Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 115

Die Peano-Kurve für Dünngitterhierarchisierung mit raumfüllenden Kurven

Marcel Schneider

Studiengang:	Informatik
Prüfer/in:	Jun.-Prof. Dr. Dirk Pflüger
Betreuer/in:	Jun.-Prof. Dr. Dirk Pflüger
Beginn am:	10. Februar 2014
Beendet am:	12. August 2014
CR-Nummer:	G.1.0, G.1.2, F.2.0

Kurzfassung

Übliche Gitteransätze zur Diskretisierung von Räumen leiden an einem exponentiellen Anstieg der Anzahl Freiheitsgrade mit der Dimensionszahl und sind deshalb nicht für höherdimensionale Probleme geeignet. Einen Ausweg bieten dünne Gitter, die deutlich weniger Freiheitsgrade benötigen. Allerdings ist die Struktur der dünnen Gitter komplexer, was praktisch zu Problemen der Cache-Ausnutzung führt: bei der Dünngittertraversierung können viele Speicherzugriffe nicht aus Cache-Speichern bedient werden.

In dieser Arbeit werden raumfüllende Kurven benutzt, um eine Operation auf dünnen Gittern, die Hierarchisierung, Cache-effizient durchzuführen. Dazu wird ein Stack & Stream Verfahren vorgestellt, das als alleinige Datenstruktur Stacks verwendet und so Cache-Effizienz garantiert, ohne das spezifische Annahmen über die Cache-Struktur gemacht werden müssen.

Inhaltsverzeichnis

Liste der Bezeichnungen

$(a,b)-(0,0)$	Volles Gitter mit maximaler Verfeinerung a in der 1. Dimension, b in der 2. Dimension
$(a,b)-(c,d)$	Rechteck im Unterraumschema, Differenz zwischen Gittern
(x,y)	Zelle im Unterraumschema bzw. hierarchisches Inkrement zur x. Verfeinerung in der 1. Dimension und y. Verfeinerung in der 2. Dimension
$\phi_{\mathbf{l},\mathbf{i}}$	Hütchenfunktion mit Level $\mathbf{l}$ und Index $\mathbf{i}$ in Zwei- oder Dreiteilung
ε	Das leere Wort, Rekursionsabbruch
${}^t\phi_{\mathbf{l},\mathbf{i}}$	Hütchenfunktion mit Level $\mathbf{l}$ und Index $\mathbf{i}$ für ein t-geteiltes Gitter
$B_{\mathbf{l}}$	Knotenbasis mit Level $\mathbf{l}$
d	Anzahl Raumdimensionen
H'_n	Dünngitterbasis mit Dünngitter-Level n
$H_{\mathbf{l}}$	Hierarchische Basis mit maximalem Level $\mathbf{l}$
k-fache Differenz	Differenz zwischen Gittern die durch Vergröbern in k verschiedenen Dimensionen entstanden ist (für $k = d$ ist dies eine einzelne Zelle im Unterraumschema)
l	Hierarchisches Level einer Ansatzfunktion oder Knotenbasis
l_{max}	Maximales Level von Ansatzfunktionen in einer hierarchischen Basis
m	Anzahl Knoten in einer Dimension für ein volles Gitter
N	Gesamtzahl Gitterpunkte/Ansatzfunktionen
n	Level eines dünnen Gitters
t	Anzahl Teilbereiche in einer rekursiven Unterteilung des Raumes
Anordnung	Abfolge der Verfeinerungsschritte einer Peanokurve als String aus Dimensionsnummern

Cache ..... Speicher in der Speicherhierarchie der vom Programmierer nicht explizit verwendet wird

Cache Miss ..... Speicherzugriff, der nicht aus einem Cache beantwortet werden kann

flache Speicherung Ein volles Gitter wird anhand einer Peanokurve auf genau einen Stack geschrieben

Freiheitsgrad .... Koeffizient für die Darstellung einer Funktion in hierarchischer oder Knotenbasis

getrennte Speicherung Ein volles Gitter wird anhand einer Peanokurve auf einen In-/Outputstack sowie weitere Transportstacks geschrieben

Gitterpunkt ..... Durch die Ansatzfunktion bestimmte Position eines Knotens im Raum

Gitterzelle ..... Zelle des maximal verfeinerten Gitters

hierarchische Zelle Zelle, die im hierarchischen Abstieg in einem Gitter auftaucht

in-place ..... Berechnung, die die Eingabe überschreibt und sublinear zusätzlichen Speicher benötigt

Knoten ..... Funktionswerte und/oder Freiheitsgrade, die einem Gitterpunkt zugeordnet sind

Reihenfolge ..... Exakte Abfolge der Knoten auf einem Stack, nicht eindeutig durch die Anordnung festgelegt

Richtung ..... Bewegungsrichtung bei einer Verfeinerung in einer Dimension (aufsteigende/absteigende Koordinatenwerte)

sortiert ..... Anordnung von Verfeinerungsschritten, in der alle Schritte in einer Dimension direkt aufeinander folgen

1 Einleitung

Für viele Anwendungen werden Funktionen mit Hilfe von Gittern approximiert. Dieser Ansatz ist jedoch für hochdimensionale Räume problematisch, da die Anzahl benötigter Gitterpunkte exponentiell mit der Dimensionszahl steigt. Deshalb wurden *dünne Gitter* eingeführt [BG04], die diesen *Fluch der Dimensionalität* umgehen. Während dünne Gitter die Anzahl benötigter Gitterpunkte deutlich senken, entsteht jedoch ein neues Problem: die Struktur dünner Gitter lässt sich nicht direkt in den eindimensionalen Speicherbereich eines Computers abbilden. Heute werden oft Hashmaps benutzt, um die benötigten Werte zu speichern, z. B. in der Implementierung SG^{++} [Pfl10]. Hashmaps erlauben jedoch Aufgrund der Randomisierung keine effektive Nutzung der Cachespeicher eines Rechners.

In dieser Arbeit werden Algorithmen vorgestellt, die die *Hierarchisierungsoperation* auf dünnen Gittern *Cache-effizient* durchführen. Dazu werden die Gitterpunkte anhand einer raumfüllenden Kurve durchlaufen, wobei alle benötigten Daten über *Stacks* transportiert werden. Solche *Stack & Stream* Algorithmen sind automatisch Cache-effizient.

Die Idee, Stack & Stream Algorithmen mit Hilfe von raumfüllenden Kurven zu entwickeln, geht zurück auf Prof. Christoph Zenger, der sowohl über meinen Betreuer Jun.-Prof. Dirk Pflüger als auch im persönlichen Gespräch viele Ansätze und Ideen für diese Arbeit beigesteuert hat.

Ein Stack & Stream Algorithmus, der auf einer Peanokurve aufbaut, wurde zum Beispiel von Frank Günther [Gün04] zum Lösen partieller Differentialgleichungen mit Hilfe der Finite-Elemente-Methode vorgestellt. Ein anderer Ansatz wurde von Gerhard Zumbusch [Zum00] vorgestellt: dort wird eine raumfüllende Kurve benutzt, um die *Knoten* eines Gitters zu traversieren, während in dieser Arbeit die *Zellen* durchlaufen werden. Anders als in dieser Arbeit war das Ziel dort eine effiziente Verteilung auf mehrere Prozessoren. Dünne Gitter auf Basis einer Dreiteilung, wie sie hier verwendet werden, wurden schon früher benutzt, z. B. von Benjamin Peherstorfer [Peh13].

Ein Algorithmus zur Hierarchisierung dünner Gitter nach dem Stack & Stream Prinzip wurde von Patrick von Steht [Ste13] vorgeschlagen. Dieser arbeitet mit der Hilbertkurve und ist dadurch auf zwei Dimensionen beschränkt. Die Einschränkungen, die ein solcher Ansatz mit sich bringt, sollen durch die Verwendung der Peanokurve in dieser Arbeit vermieden werden.

Gliederung

Die Arbeit ist in wie folgt gegliedert:

Kapitel 2 – Grundlagen: Hier werden kurz die Grundlagen zu dünnen Gittern eingeführt.

Kapitel 3 – Cache-Effiziente Algorithmen führt das Konzept der Cachespeicher und Entwurfsansätze für Cache-effiziente Algorithmen ein.

Kapitel 4 – Raumfüllende Kurven: In diesem Kapitel wird die Konstruktion von raumfüllenden Kurven, vor allem der hier benutzten Peanokurve, und ihre Verwendung zum Traversieren von Gittern eingeführt.

Kapitel 5 – Hierarchisierungsalgorithmen für volle Gitter erläutert verschiedene Hierarchisierungsalgorithmen für volle Gitter.

Kapitel 6 – Stack & Stream Algorithmus für dünne Gitter: In diesem Kapitel wird schließlich aus den vorher eingeführten Ansätzen ein Cache-effizienter Hierarchisierungsalgorithmus für dünne Gitter konstruiert.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

Im Folgenden sollen kurz die zum Verständnis der später eingeführten Algorithmen nötigen Grundlagen eingeführt werden. Den Prinzipien Cache-effizienter Algorithmen und raumfüllender Kurven werden eigene Kapitel gewidmet; dieses Kapitel bespricht deshalb vor allem die Grundlagen hierarchischer und dünner Gitter.

2.1 Hierarchische Basis

Im Kontext dieser Arbeit werden *Gitter* benutzt, um Funktionen $\mathbb{R}^d \to \mathbb{R}$ zu approximieren. Dazu werden mit den in räumlich gleichmäßigen Abständen angeordneten *Knoten* Ansatzfunktionen ϕ assoziiert, als deren gewichtete Summe sich eine Näherung an die gesuchte Funktion ergibt:

$$f(\mathbf{x}) \approx f_N(\mathbf{x}) := \sum_i a_i \phi_i(\mathbf{x}) \tag{2.1}$$

Die N Ansatzfunktionen ϕ_i bilden zusammen die Basis eines Funktionsraumes. Jede Funktion aus diesem Funktionsraum wird eindeutig bestimmt durch die N *Freiheitsgrade* a_i.

2.1.1 Ansatzfunktionen

Als Ansatzfunktionen werden stückweise lineare *Hütchenfunktionen* benutzt. Diese Funktionen $\phi_{l,i}$ werden charakterisiert durch ihr *Level* l, das die Ausdehnung der Funktion beschreibt, und den *Index* i, der die räumliche Position angibt. Eindimensional werden die Funktionen wie folgt definiert:

$$\phi_{l,i}(x) := \max\left\{1 - \left|2^l x - i\right|, 0\right\} . \tag{2.2}$$

Mehrdimensionale Varianten davon werden mit Hilfe des Tensorproduktansatzes als Produkt von eindimensionalen Funktionen konstruiert:

$$\phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) := \prod_j \phi_{l_j,i_j}(x_j) . \tag{2.3}$$

Dabei sind $\mathbf{l}$ und $\mathbf{i}$ Multiindices mit d Einträgen für d Dimensionen.

Mit Hilfe dieser Ansatzfunktionen lassen sich verschiedene sinnvolle Basen konstruieren. Die einfachste Möglichkeit ist die *Knotenbasis*, die den Raum $V_\mathbf{l}$ aus stückweise linearen Funktionen aufspannt. Wir betrachten dabei nur den Einheitswürfel $[0,1]^d$. $V_\mathbf{l}$ enthält dann die Funktionen, die zwischen den 2^{l_j} Gitterpunkten in jeder Dimension j d-linear sind. Die

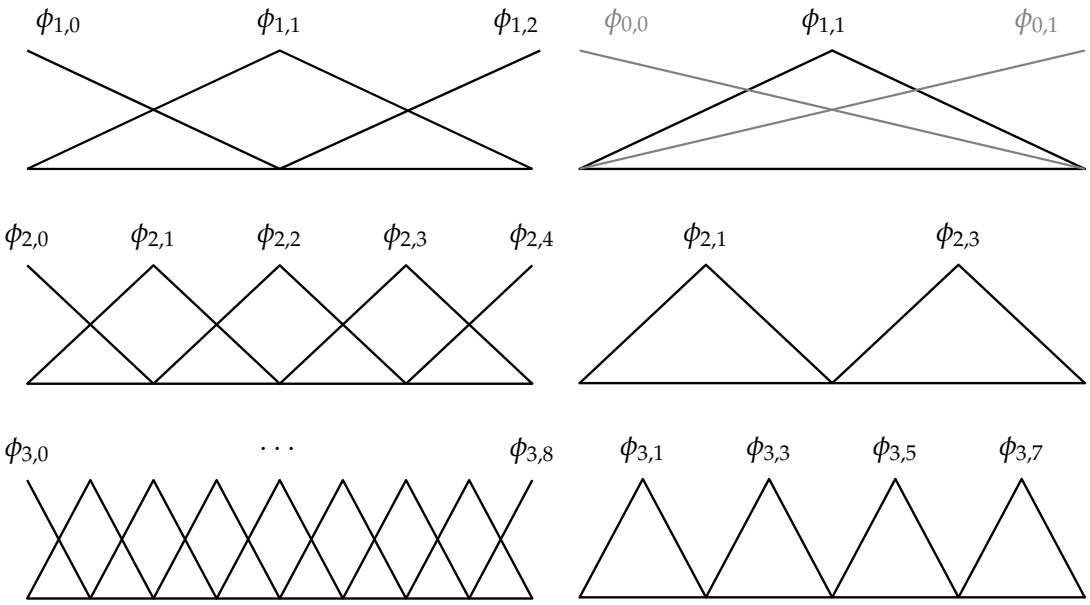

Abbildung 2.1: Knotenbasen vom Level 1 bis 3 und hierarchische Basis Level 3 in einer Dimension im Vergleich. Auf Level 1 der hierarchischen Basis wurden zwei Ansatzfunktionen vom Level 0 ergänzt, um den Rand ebenfalls abzudecken.

Knotenbasis $B_\mathbf{l}$ für diesen Raum enthält dann Ansatzfunktionen ϕ vom Level $\mathbf{l}$, und zwar die mit Index $\mathbf{i}$ aus der Indexmenge

$$I_k := \left\{1, \ldots, 2^{l_k} - 1\right\} \tag{2.4}$$

$$I_\mathbf{l} := I_{l_1} \times \cdots \times I_{l_d}\,. \tag{2.5}$$

An jedem Gitterpunkt $\mathbf{x} = \mathbf{i} \cdot 2^{-\mathbf{l}}$, alle Operationen elementweise, ist genau die Ansatzfunktion $\phi_{\mathbf{l},\mathbf{i}}$ 1, während alle anderen Funktionen in $B_\mathbf{l}$ den Wert 0 haben.

2.1.2 Hierarchisierung

Eine andere Basis für den Funktionsraum $V_\mathbf{l}$ ist die *hierarchische* Basis $H_\mathbf{l}$. Hier wird die Eigenschaft, dass an jedem Gitterpunkt nur eine Ansatzfunktion ungleich 0 ist, sowie die Beschränkung auf Ansatzfunktionen mit Level $\mathbf{l}$ aufgegeben, und stattdessen ein hierarchischer Ansatz gewählt.

Die hierarchische Basis enthält zunächst die Ansatzfunktion $\phi_{\mathbf{1},\mathbf{1}}$, die den gesamten Einheitswürfel überspannt. Für jeden linearen Bereich dieser Funktion werden dann rekursiv Ansatzfunktionen vom nächsthöheren Level hinzugefügt, die genau diese Bereiche überspannen, bis das gewünschte Level erreicht ist.

Nach dieser Konstruktion ergibt sich

$$H_{\mathbf{l}} := \left\{ \phi_{\mathbf{k},\mathbf{i}} \mid \forall_j \, 0 < k_j \leq l_j, 0 < i_j < 2^{l_j}, i_j \text{ ungerade} \right\} . \tag{2.6}$$

Da nun $H_{\mathbf{l}}$ und $B_{\mathbf{l}}$ den selben Raum $V_{\mathbf{l}}$ aufspannen [BG04], lässt sich jede Funktion aus $V_{\mathbf{l}}$ in beiden Basen darstellen, wobei beide Darstellungen eindeutig sind. Die Transformation von der Knotenbasis in die hierarchische Basis wird als *Hierarchisierung* bezeichnet, die umgekehrte Transformation als *Dehierarchisierung*. Diese beiden Transformationen sollen in dieser Arbeit näher betrachtet werden. Dabei wird meist nur von der Hierarchisierung gesprochen, die meisten Verfahren lassen sich aber für beides Anwenden.

2.2 Dünne Gitter

Ein Problem der oben eingeführten *vollen* Gitter ist der *Fluch der Dimensionalität*: die Anzahl Freiheitsgrade für ein Gitter mit m Knoten in jeder der d Dimensionen liegt in $\mathcal{O}(m^d)$. Für mehr als drei Dimensionen sind solche Verfahren deswegen praktisch nicht mehr einsetzbar. *Dünne* Gitter lösen dieses Problem teilweise: ein dünnes Gitter besitzt bei vergleichbarer Auflösung nur $\mathcal{O}(m(\log m)^{d-1})$ Freiheitsgrade [BG04]. Offensichtlich kann ein solches dünnes Gitter nicht alle Funktionen aus $V_{\mathbf{l}}$ darstellen, der Approximationsfehler bleibt aber gering.

Ausgehend von der hierarchischen Basis lassen sich dünne Gitter einfach dadurch definieren, dass Ansatzfunktionen mit sehr hohem Level weggelassen werden:

$$H'_n := \{ \phi_{\mathbf{l},\mathbf{i}} \in H_* \mid |\mathbf{l}|_1 \leq n + d - 1 \} . \tag{2.7}$$

Dabei ist die $|.|_1$-Norm eines Multiindex definiert als

$$|\mathbf{l}|_1 := \sum_j l_j \tag{2.8}$$

und H_* ist die hierarchische Basis mit beliebig hohem Level.

Eine Knotenbasis lässt sich für dünne Gitter nicht definieren. Es ist allerdings möglich, eine Zerlegung des dünnen Gitters in volle Gitter mit Level $\mathbf{l}$ so wie in (2.7) zu betrachten. Die Vereinigung der Knotenbasen bildet dann ein Erzeugendensystem:

$$B'_n := \bigcup_{\substack{|\mathbf{l}|_1 \leq n+d-1 \\ \mathbf{l} \geq \mathbf{1}}} B_{\mathbf{l}} . \tag{2.9}$$

Dieses enthält mehr Ansatzfunktion als die hierarchische Basis und ist deshalb keine Basis, außerdem gilt die Eigenschaft, dass an jedem Gitterpunkt nur eine Ansatzfunktion ungleich 0 ist, nicht mehr.

Die Hierarchisierung ist auf dünnen Gittern *keine* Basistransformation. Stattdessen werden die Koeffizienten für die hierarchische Basis des dünnen Gitters aus Funktionsauswertungen

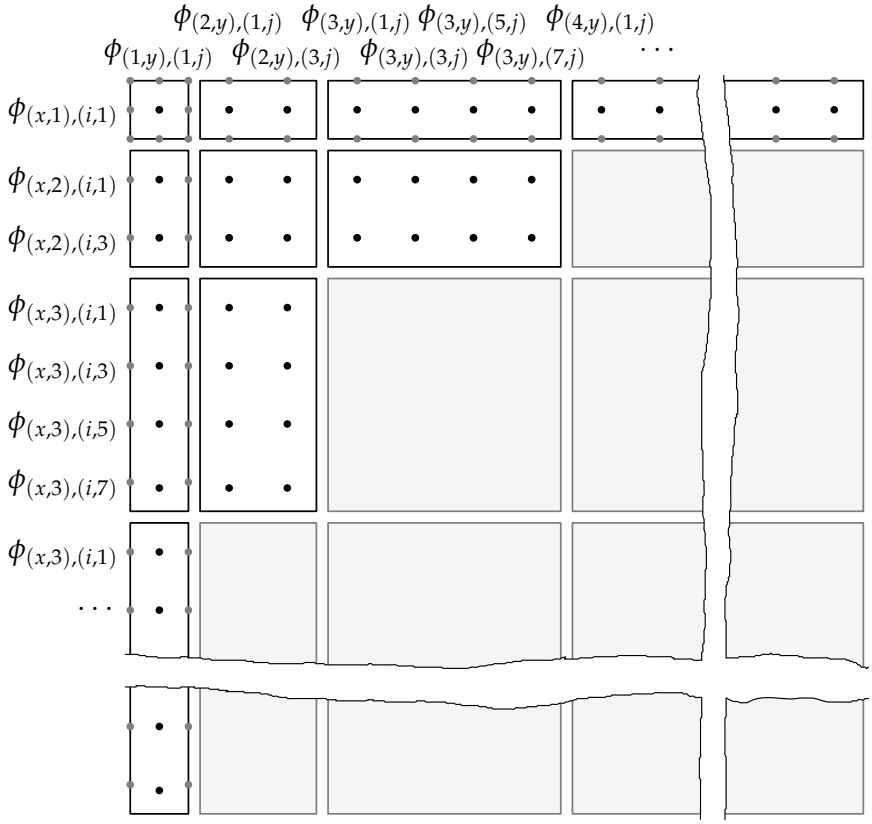

Abbildung 2.2: Die im dünnen Gitter benutzten und eingesparten (grau) Gitter. Die Fläche im Diagramm entspricht der Anzahl Freiheitsgrade; obwohl nur etwa die Hälfte der Gitter weggelassen wird, sinkt die Anzahl Freiheitsgrade stark.

an den Gitterpunkten bestimmt, es gibt aber keine Basis in der diese Auswertungen direkt die Freiheitsgrade der Darstellung der gesuchten Funktion sind.

Alle bisher eingeführten Gitter enthalten noch keine Ansatzfunktionen, die auf dem Rand des Definitionsbereichs ungleich 0 sind. Um auch Funktionen, die am Rand nicht 0 sind darstellen zu können, können verschiedene Erweiterungen vorgenommen werden. Eine Möglichkeit für dünne Gitter ist es, auf Level 1 in jeder Dimension zwei zusätzliche Basisfunktionen vom Level 0 hinzuzufügen, die auf dem Rand zentriert sind und den gesamten Bereich überspannen (Abb. 2.1). Für eine detailliertere Beschreibung dünner Gitter siehe [BG04].

2.3 Dreiteilung

Bisher wurden klassische hierarchische und dünne Gitter, die auf einer rekursiven Aufteilung in je zwei Teilbereiche basieren, vorgestellt. Um später Algorithmen anwenden zu können, die auf Peanokurven basieren, ist eine analoge Konstruktion mit je drei Teilbereichen nötig. Formal werden hier beliebige t-geteilte hierarchische Gitter definiert, allerdings werden im

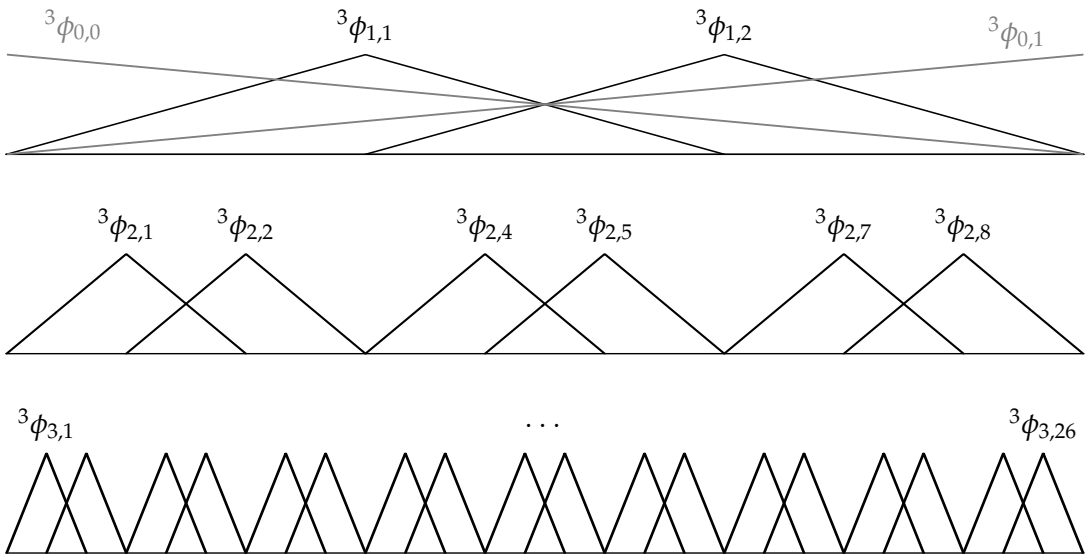

Abbildung 2.3: Hierarchische Basis mit Dreiteilung. Jedes Intervall wird durch zwei Freiheitsgrade in drei Teilintervalle zerlegt.

Folgenden nur 2- *oder* 3-geteilte Gitter verwendet, so dass der vorangestellte Index ^{2}. oder ^{3}. in der Regel nicht angegeben werden braucht.

Die Form der ϕ bleibt an sich unverändert, jedoch wird die Bedeutung des Parameters $\mathbf{l}$ angepasst. Anstatt Zweierpotenzen bezieht sich das Level nun auf Dreierpotenzen, oder allgemeiner

$$^t\phi_{l,i}(x) := \max\left\{1 - \left|t^l x - i\right|, 0\right\} \tag{2.10}$$

für t-geteilte Gitter.

Die Knotenbasis ergibt sich ebenfalls durch Austauschen der Potenzen als

$$^tI_k := \left\{1, \ldots, t^k\right\} \tag{2.11}$$

$$^tB_{\mathbf{l}} := \left\{^t\phi_{\mathbf{l},\mathbf{i}} \mid \mathbf{i} \in {}^tI_{l_1} \times \cdots \times {}^tI_{l_d}\right\} . \tag{2.12}$$

Für die hierarchische Basis wird nun nicht jede zweite, sondern jede dritte Ansatzfunktion eines Levels durch eine gröberen Levels ersetzt:

$$^tH_{\mathbf{l}} := \left\{^t\phi_{\mathbf{k},\mathbf{i}} \mid \forall_j\, k_j \leq l_j, 0 < i_j < t^{l_j}, i_j \not\equiv 0 \mod t\right\} . \tag{2.13}$$

Die entsprechenden Dünngitter-Basen $^3H'_n$ ergeben sich genau wie die zweigeteilten Varianten aus den Basen für allgemeines t.

3 Cache-Effiziente Algorithmen

In dieser Arbeit soll es vor allem um Cache-effiziente Algorithmen gehen, also Algorithmen, die die Cachespeicher eines Computersystems besonders gut ausnutzen. Dazu wird im Folgenden zunächst das Konzept des *Caches* eingeführt und verschiedene Ansätze zur Konstruktion von Cache-effizienten Algorithmen vorgestellt.

3.1 Speicherhierarchie

Ein einfaches Berechnungsmodell, für das Algorithmen entworfen und analysiert werden können, ist die *Registermaschine* (eng. Register Access Machine, *RAM*). Diese Maschine besteht neben einem endlichen Steuerwerk, welches einem Programm entspricht, aus einer unendlichen Anzahl Registern, die je eine Zahl, speichern können.

Auf jedes Register kann, ggf. auch per Referenz aus einem anderen Register, in konstanter Zeit zugegriffen werden. Dieses Modell entspricht soweit dem Verhalten der ersten praktisch realisierten Rechner. Schon seit längerer Zeit zeigen reale Maschinen aber ein anderes Verhalten: Die Kosten eines Speicherzugriffes sind nicht konstant.

Während die Rechengeschwindigkeit von Prozessoren in den letzten Jahren permanent gestiegen ist, hat sich die Zugriffsgeschwindigkeit auf Speicher langsamer entwickelt. Die Geschwindigkeit des Speichers wurde so zum Flaschenhals [HP12]. Praktisch wurde dieses Problem durch eine Aufteilung des Speichers in eine *Speicherhierarchie* gelöst: Häufig benötigte Werte liegen in kleinen, sehr schnellen Speichern, während seltener benötigte Werte in größeren, aber langsameren Speichern abgelegt werden.

Die Speicherhierarchie kann aus vielen Komponenten bestehen: neben den Speichern, die explizit unterschieden werden, wie CPU-Registern, Hauptspeicher und Festplattenspeicher befinden sich dort *Caches*, die hier näher betrachtet werden.

3.1.1 Caches

Als *Cachespeicher* oder *Caches* werden in dieser Arbeit Speicher in der Speicherhierarchie bezeichnet, auf deren Verwendung der Programmierer keinen expliziten Einfluss hat. Am bedeutendsten sind hier die als $L1$, $L2$, usw. bezeichneten Caches zwischen CPU und Hauptspeicher. Ähnlich verhält sich aber auch der Dateisystem-Cache eines Betriebssystems, der sich zwar in einem explizit adressierbaren Bereich des Hauptspeichers befindet, aber für den Programmierer transparent vom Betriebssystem verwaltet wird.

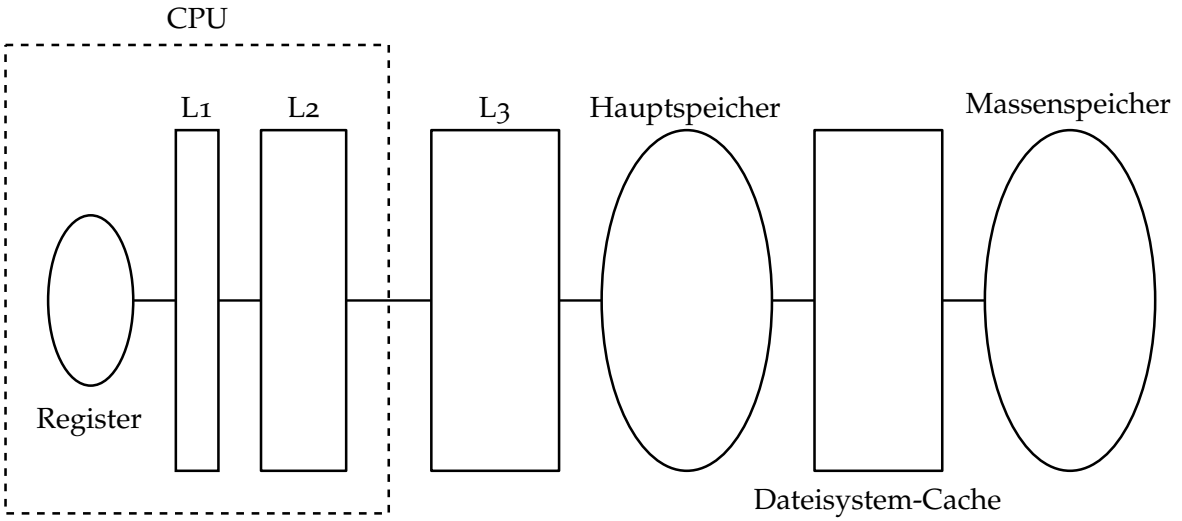

Abbildung 3.1: Speicherhierarchie mit verschiedenen, explizit getrennten Schichten (Hauptspeicher, Festplattenspeicher) und Caches, die aus Anwendungssicht transparent sind.

Welche Werte genau in Caches vorgehalten werden können, und welche aus tieferen Schichten der Hierarchie geladen werden müssen, hängt von verschiedenen Parametern ab. Im Fall der Hardware-Caches zwischen CPU und Hauptspeicher findet die Verwaltung komplett innerhalb des Prozessors statt. Die genauen Strategien zur Einlagerung neuer und Verdrängung nicht mehr gebrauchter Werte aus dem Cache werden vom Hersteller festgelegt und sind oft nicht genau bekannt, da entsprechende Details nicht veröffentlicht werden oder sich mit neuen Generationen schnell verändern.

3.1.2 Lokalität

Da das exakte Verhalten eines Caches nicht bekannt ist, ist eine allgemeinere Beschreibung nötig. Allen Caches gemeinsam ist das Konzept der *Lokalität*, im Raum (hier zu verstehen als abstrakter Adressraum) und in der Zeit.

- Das Prinzip räumlicher Lokalität besagt, dass wenn auf eine Speicherstelle zugegriffen wird, die nächsten Zugriffe wahrscheinlich auf Adressen mit einem geringen Abstand erfolgen.
- Zeitliche Lokalität besagt, dass wenn vor kurzer Zeit auf eine Speicherstelle zugegriffen wurde, wahrscheinlich noch einmal auf die selbe Stelle zugegriffen wird.

Diese Prinzipien gelten insbesondere für Programmcode, wo sich meist eine sehr starke Lokalität einstellt; als Faustregel gilt, dass 90% der Laufzeit eines Programms in nur 10% des Programmcodes stattfinden [HP12]. Diese Prinzipien werden aber auch für den Entwurf von Datencaches zugrunde gelegt, da sie auch dort meist gelten. Das Ziel Cache-effizienter Algorithmen ist es nun, auf Daten so zuzugreifen, dass sich eine möglichst starke Lokalität

einstellt, so dass die in der Hardware implementierten Strategien gut funktionieren und möglichst viele Zugriffe aus den Caches bedient werden können.

Prinzipiell ist es möglich, mit Kenntnis der Struktur und benutzten Strategien eines Caches, Algorithmen zu entwerfen, die mit diesem Cache besonders gut funktionieren (*Cache aware algorithm*). Da solche Informationen aber meist nicht vorhanden sind, und solche Algorithmen sehr unflexibel sind, beschränkt sich diese Arbeit vor allem auf Algorithmen die *ohne* Kenntnis der Cache-Struktur auskommen (*Cache oblivious algorithm*) [FLPR99].

Als Maß für den Erfolg einer effizienten Cache-Nutzung wird die Anzahl Speicherzugriffe, die aus dem Cache bedient werden können (*Cache Hits*) und die Zahl der Zugriffe, für die dies nicht gelingt (*Cache Misses*) betrachtet. Das Verhältnis zwischen diesen Zahlen ist ein Maß für die Cache-Effizienz. Dieses hängt aber immer vom betrachteten Cache ab; die Zahlen sind also nicht unbedingt vergleichbar zwischen verschiedenen Maschinen.

3.2 Divide & Conquer

Ein möglicher Ansatz zur Konstruktion Cache-effizienter Algorithmen ist es, das gegebene Problem rekursiv in kleinere Teilprobleme zu zerlegen. Diese Teilprobleme erreichen irgendwann eine Größe, die in den Cache passt, und können effizient bearbeitet werden. Entscheidend ist, dass die Teilprobleme im Speicher zusammenhängend sind und für das Aufteilen und des Problems und Zusammensetzen der Lösung keine weiteren Speicherzugriffe nötig sind.

3.2.1 Matrix Transpose

Einige Algorithmen nach diesem Ansatz wurden von Harald Prokop [FLPR99] eingeführt. Hier soll beispielhaft der Algorithmus für Cache-effizientes transponieren einer Matrix vorgestellt werden, da dieser später als Teil eines Hierarchisierungsalgorithmus für volle Gitter genutzt werden kann.

Wir beschränken uns auf quadratische Matrizen mit Größen, die Zweier- oder Dreierpotenzen sind. Der Algorithmus wird hier der Einfachheit halber nur für Zweierpotenzen eingeführt. Als weitere Anforderung soll die gegebene Matrix jedoch *in-place* transponiert werden, d. h. das Ergebnis ersetzt im Speicher die Eingabe und es wird kein weiterer Platz benötigt. Das Problem kann also wie folgt formuliert werden: Gegeben eine Matrix A der Form $2^n \times 2^n$ in zeilenweiser Anordnung, konvertiere A in eine spaltenweise Anordnung ($= A^T$ in zeilenweiser Anordnung).

Der Algorithmus arbeitet intern dennoch mit zwei Matrizen A und B, die allerdings am Anfang identisch sind; im Laufe der rekursiven Zerlegung können sie aber verschieden sein. Es muss jedoch besonders darauf geachtet werden, dass der Algorithmus in beiden Fällen korrekt ist. Der Algorithmus besteht aus einem Basisfall (Half-Transpose$(A, B, diag)$) der naiv die obere Dreiecksmatrix von A transponiert mit der unteren Dreiecksmatrix von B vertauscht.

Algorithmus 3.1 Rekursives transponieren einer Matrix

```
procedure Rec-Transpose(A, B, diag)
    if dim(A) < c then
        Half-Transpose(A, B, diag)
    else
        Rec-Transpose(A_2, B_3, true)
        Rec-Transpose(A_3, B_2, false)
        Rec-Transpose(A_1, B_1, diag)
        Rec-Transpose(A_4, B_4, diag)
    end if
end procedure
```

Abhängig vom Parameter *diag* wird die Diagonale dabei ebenfalls getauscht oder ausgelassen. Insbesondere transponiert Half-Transpose($A, A, false$) oder Half-Transpose($A, A, true$) die Matrix A in-place. Die Prozedur Rec-Transpose($A, B, diag$) hat die selbe Wirkung, ruft aber rekursiv sich selbst bzw. Half-Transpose auf.

Die Prozedur Rec-Transpose betrachtet folgende Zerlegung:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \tag{3.1}$$

Um das gewünschte Ergebnis zu erreichen, müssen nun A_2^T und A_3^T mit B_3 und B_2 vertauscht werden, außerdem A_1^T und A_4^T mit B_1 und B_4. Dies erledigt Algorithmus 3.1 mit vier rekursiven Aufrufen bzw. Aufrufen von Half-Transpose. Der Parameter *diag* wird dabei benötigt, um sicherzustellen, dass die Diagonaleinträge genau einmal vertauscht werden. Das ursprüngliche Problem löst nun Rec-Transpose(A) := Rec-Transpose($A, A, true$).

Die Cache-Effizienz dieses Algorithmus besteht darin, dass die Prozedur Rec-Transpose selbst keine Speicherzugriffe durchführt. Die Matrix-Parameter A und B haben Referenz-Semantik, so dass das Zerlegen der Matrizen nur durch anpassen der Indexgrenzen erreicht wird. Nach einer gewissen Anzahl Zerlegungen passen die Matrizen A und B, tatsächlich also zwei Teile der ursprünglichen Matrix A, in den Cache, und die Reihenfolge der Speicherzugriffe durch Half-Transpose ist unbedeutend. Deshalb ist auch der Wert des Parameters c unkritisch: die Rekursion *darf* erst enden, wenn beide Matrizen in den Cache passen; sie *kann* aber das Problem auch noch weiter unterteilen, ohne das dies Nachteile für die Cache-Effizienz bringt. Aus Sicht der Cache-Effizienz muss c also nur ausreichend klein sein, praktisch sollte c aber so groß gewählt werden, dass die die Teilprobleme gerade noch in den Cache passen, um die teuren rekursiven Aufrufe einzusparen.

3.3 Stack & Stream

Ein anderer Ansatz zum Entwurf Cache-effizienter Algorithmen ist die Verwendung von *Stacks* bzw. *Kellerspeichern* als alleinige Datenstruktur. Ein Stack ist eine Datenstruktur, auf

der nur die beiden Operationen *push* und *pop* erlaubt sind. Die Operation *push* legt ein neues Element auf den Stack; die Operation *pop* liefert den Wert zurück, der zuletzt mit *push* geschrieben wurde, und entfernt ihn vom Stack. Zudem muss ein Programm erkennen können, ob der Stack leer ist, dies wird durch das Prädikat *empty* angezeigt. Auf einem leeren Stack darf kein *pop* ausgeführt werden.

Stacks können leicht auf den eindimensionalen Adressbereich einer Registermaschine abgebildet werden. Dazu wird ein *Stackpointer* benutzt, der immer auf die letzte genutzte (alternativ erste freie) Zelle zeigt. Für die *push*-Operation wird dann der Stackpointer inkrementiert (alternativ dekrementiert) und der Wert an die Stelle geschrieben, für *pop* wird der Wert, auf den der Stackpointer zeigt zurückgegeben und der Stackpointer dekrementiert.

Ein so dargestellter Stack ist inhärent Cache-effizient: die nächste gelesene oder geschriebene Zelle ist immer direkt benachbart zur vorherigen, die räumliche Lokalität ist also immer gegeben. Zeitliche Lokalität ist ebenfalls gegeben: es werden immer die Elemente gelesen, die zuletzt geschrieben wurden.

Für spätere Algorithmen wird zudem erlaubt, neben dem aktuell obersten Element eines Stacks auch auf eine konstante Anzahl darunterliegender Elemente zuzugreifen. Aus theoretischer Sicht ist dies auch mit den normalen Stack-Operationen möglich, da auf dem Stack jeweils Tupel der letzten k Elemente abgelegt werden könnten, so dass immer die k letzten Elemente zugleich sichtbar sind. Aus praktischer Sicht ist dies ebenfalls unproblematisch, da die Elemente direkt benachbart zum obersten Element liegen. Natürlich darf k nicht zu groß sein, damit die Elemente noch im Cache zu finden sind.

Ein *Stack & Stream*-Algorithmus lagert alle Daten auf Stacks. Die Eingabe wird von einem *Inputstack* gelesen, die Ausgabe auf einen *Outputstack* geschrieben. Während der Verarbeitung können Daten auf einer konstanten Anzahl weiterer Stacks gelagert werden. Wenn die Anzahl Elemente auf In- und Outputstack zusammen nie größer als die Eingabe wird, können Stack & Stream-Algorithmen auch *in-place* arbeiten, indem der Outputstack den Speicherbereich des Inputstack nutzt, aber in die entgegengesetzte Richtung wächst. Für die weiteren Stacks wird natürlich zusätzlicher Speicher benötigt, die Menge hängt vom Algorithmus ab.

3.3.1 Kellerautomaten

Als Spezialfall von Stack & Stream Algorithmen können *Kellerautomaten* aufgefasst werden, ein Berechnungsmodell aus der theoretischen Informatik, das auch im Compilerbau Anwendung findet. Ein Kellerautomat ist eine Maschine, die neben einem endlichen Steuerwerk genau einen Stack als Speicher für Berechnungen nutzt. Die Eingabe kommt von einem *Eingabeband*, das als Inputstack aufgefasst werden kann, mit der Einschränkung, das kein *push* darauf erlaubt ist. Eine Ausgabe gibt es nicht, es wird nur akzeptiert oder nicht akzeptiert.

Kellerautomaten sind interessant, weil sie zeigen, dass die Einschränkung auf die Stack-Operationen eine echte Einschränkung ist: obwohl Kellerautomaten einen prinzipiell unendlichen Speicher zur Verfügung haben, sind sie weniger mächtig als Turingmaschinen. Diese Einschränkung löst sich, sobald mehr als ein Stack erlaubt wird. Interessant ist auch, dass

ein direkter Zusammenhang zwischen Kellerautomaten und kontextfreien Grammatiken besteht.

Eine Sprache kann genau dann von einem (möglicherweise nichtdeterministischen) Kellerautomat erkannt werden, wenn sie von einer kontextfreien Grammatik erzeugt wird. Für die praktische Anwendung besonders nützlich ist, dass für jede kontextfreie Sprache ein Kellerautomat konstruiert werden kann, der in jedem Rechenschritt ein Zeichen der Eingabe liest, der also Worte in Linearzeit verarbeitet. Dies wird erreicht, in dem der Kellerautomat aus der *Greibach-Normalform* einer kontextfreien Grammatik konstruiert wird [Scho8].

Für die Anwendung z. B. im Compilerbau ungünstig sind zwei Dinge: einerseits der Nichtdeterminismus des Automaten, der auf Rechnerhardware nur mit großem Aufwand simuliert werden kann, andererseits die Umformung in die Greibach-Normalform, bei der die Struktur der Grammatik verändert wird. In praktischen Anwendungen ist man meist nicht nur an der Entscheidung des Wortproblems interessiert, die in der theoretischen Informatik untersucht wird, sondern an einer Ableitung eines Wortes aus einer gegebenen Grammatik, die nur sehr begrenzt verändert werden darf. Beide Probleme wurden für die Anwendung im Compilerbau gelöst: Die Klasse der *deterministisch kontextfreien Sprachen*, eine echte Teilmenge der kontextfreien Sprachen, kann von deterministischen Kellerautomaten erkannt werden. Die meisten praktisch relevanten Sprachen, z. B. Programmiersprachen, sind deterministisch kontextfrei. Um die Umformung in Greibach-Normalform zu vermeiden, werden Grammatiken direkt in einer Form formuliert, die mehr Freiheit bietet, aber für die sich ohne weitere Umformung deterministische Kellerautomaten, die die Eingabe in Linearzeit verarbeiten, konstruieren lassen.

Die bekanntesten Klassen solcher Grammatiken sind $LL(k)$ und $LR(k)$ Grammatiken [ASU86]. Die Grammatiken unterscheiden sich in der Art und Weise, in der der Stack genutzt wird. $LL(k)$-Grammatiken können durch rekursiven Abstieg geparst werden, wobei alle Entscheidungen durch Betrachten der maximal k nächsten Zeichen der Eingabe getroffen werden. Bei einem LL-Parser wird der Stack also als eine Art Aufrufstack genutzt, der immer Informationen über die zuletzt erfolgten Ableitungen enthält. LL-Grammatiken sind allerdings relativ stark eingeschränkt. Für die mächtigeren LR-Grammatiken werden Kellerautomaten konstruiert, die auf dem Keller die eingelesenen, aber noch nicht verarbeiten Zeichen einer *Satzform*, also einer unvollständigen Ableitung aus der Grammatik, speichern.

Diese Verfahren können nicht direkt auf die dieser Arbeit betrachteten Probleme angewendet werden. Die Grundideen dieser Konstruktionen können aber einen Hinweis darauf geben, wie sich Stack & Stream-Algorithmen konstruieren lassen, wenn das Problem mit Hilfe einer kontextfreien Grammatik ausgedrückt werden kann.

4 Raumfüllende Kurven

Bei der Implementierung von Gitter-basierenden Verfahren stellt sich das Problem, dass die im Allgemeinen mehrdimensionale Struktur des Gitters in den eindimensionalen Adressraum des Speichers abgebildet werden muss. Ein einfacher und verbreiteter Ansatz ist es, ein volles Gitter *zeilenweise* im Speicher abzulegen, d. h. die Elemente liegen in der Reihenfolge im Speicher, in der man sie beim zeilenweisen Ablesen aus dem Gitter, im zweidimensionalen analog zum zeilenweisen Lesen einer Seite Text, trifft. Um diese Positionen zu bestimmen, muss der Indexbereich für jede Dimension bekannt sein. Der eindimensionale Speicherindex p für ein Element $\mathbf{i}$ ergibt sich dann als

$$p = i_1 + m_1(i_2 + m_2(i_3 + \dots))) \tag{4.1}$$

wobei $\mathbf{m}$ die Anzahl Einträge pro Dimension angibt und $\mathbf{i}$ von $\mathbf{0}$ bis $\mathbf{m} - \mathbf{1}$ läuft. Je nach Reihenfolge, in der die Dimensionen durchlaufen werden, spricht man von einer *row-major* (wie hier) oder *column-major* (umgekehrte Reihenfolge der Dimensionen) Anordnung.

Ein Nachteil dieser Anordnung ist die schlechte Cache-Effizienz: Das Prinzip der räumlichen Lokalität wird beim Wechsel der Zeile, oder allgemein bei jeder Änderung in einer Dimension außer der ersten, verletzt. Hier schaffen *raumfüllende Kurven* Abhilfe: Eine raumfüllende Kurve bildet jeden zusammenhängenden Speicherindexbereich auf einen zusammenhängenden Bereich im Raum ab, und umgekehrt auch zusammenhängende Raumbereiche auf weitgehend zusammenhängende Indexbereiche.

Mathematisch definieren wir eine *Kurve* als Abbildung f aus einem Intervall $\mathcal{I} \subset \mathbb{R}$ nach $\mathbb{R}^d$ [Bad13]. Eine Kurve wird dann als *raumfüllend* bezeichnet, wenn ihr Bild $f(\mathcal{I})$ ein Jordan-Volumen größer 0 besitzt. Spezifischer werden im Folgenden nur raumfüllende Kurven betrachtet, die für $\mathcal{I} = [0, 1]$ den Einheitswürfel $\mathcal{I}^d = [0, 1]^d$ füllen, also surjektive Abbildungen von $\mathcal{I}$ nach $\mathcal{I}^d$.

Für die praktische Anwendung werden diskrete Annäherungen an eine raumfüllende Kurve betrachtet, die die gesuchte Abbildung von $\{1, \dots, m\}^d$ auf $\{1, \dots, m^d\}$ liefern. In diesem Fall liegt sogar eine bijektive Abbildung vor; im Kontinuierlichen sind keine bijektiven Abbildungen möglich.

4.1 Die Peanokurve

Die einzige raumfüllende Kurve, die im Folgenden näher betrachtet wird, ist die *Peanokurve*. Ihre Konstruktion geht zurück auf Giuseppe Peano aus dem Jahr 1890, es handelt sich dabei um die erste bekannte raumfüllende Kurve.

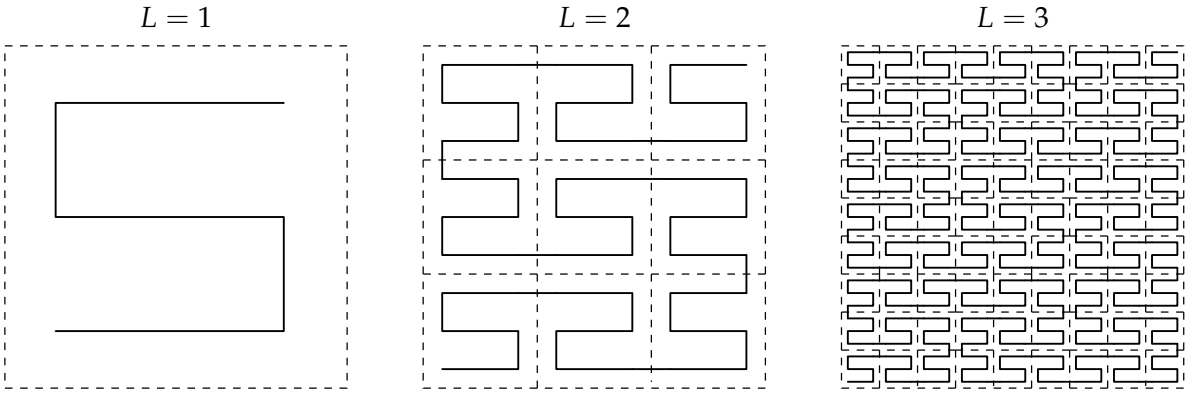

Abbildung 4.1: Die ersten drei Verfeinerungen der üblichen Peanokurve.

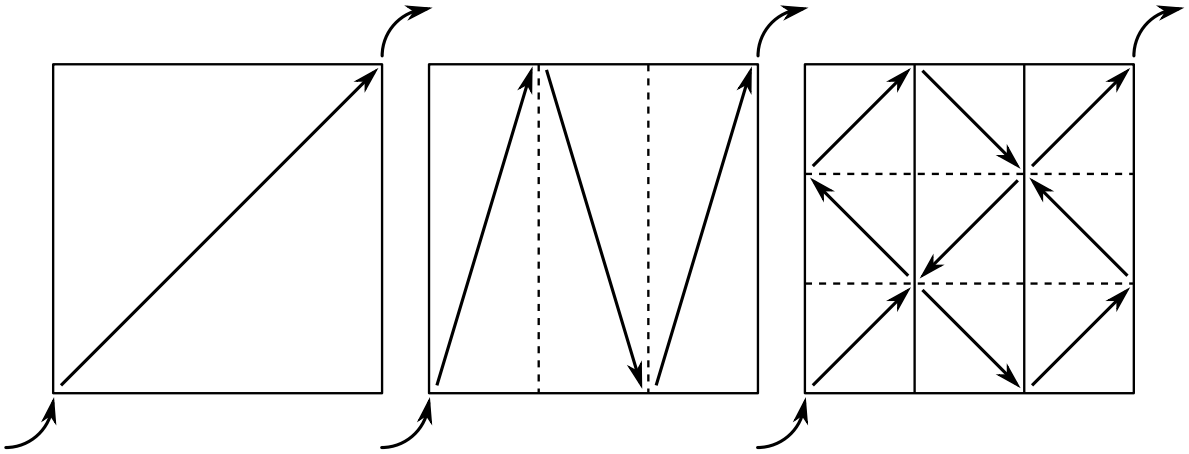

Abbildung 4.2: Die Ein- und Austrittspunkte bleiben bei weiterer Verfeinerung gleich, es ergibt sich eine regelmäßige Struktur.

Wie die meisten raumfüllenden Kurven wird die Peanokurve rekursiv definiert, wobei sich die eigentliche raumfüllende Kurve, die tatsächlich ein Jordan-Volumen größer 0 hat, als Grenzfall für eine unendliche Rekursionstiefe ergibt. Für die hier benutzten diskreten Annäherungen wird die Rekursion nach einer endlichen Anzahl Schritten abgebrochen, sobald die Kurve alle gewünschten Gitterzellen durchläuft.

Die Peanokurve basiert auf einer rekursiven Aufteilung des Raumes in jeder Dimension in je drei gleiche Teile. Für zwei Dimensionen wird das Einheitsquadrat also in 9 gleiche Teilquadrate zerlegt, die jeweils rekursiv mit einer Peanokurve gefüllt werden. Diese neun Teile werden so angeordnet, dass sich keine Sprünge über benachbarte Zellen hinweg ergeben. Durch geeignete Spiegelungen der Kurven in den Teilquadraten ergibt sich eine Kurve komplett ohne Sprünge. Dabei sind viele Lösungen möglich, von denen hier nur eine verbreitete, wie in Abb. 4.1 dargestellt, betrachtet wird.

Für die Algorithmen wird jedoch nicht die klassische, quadratisch verfeinerte Peanokurve benutzt, sondern eine Variante, die auch rechteckige Teilbereiche zulässt. Dazu werden

Unterteilungen in drei Teilbereiche für jede Dimension unabhängig betrachtet. In jedem Verfeinerungsschritt wird eine Dimension gewählt, in der die aktuellen Bereiche in drei Teile zerlegt werden. Wählt man abwechselnd die x- und y-Dimension, so ergibt sich genau die klassische Peanokurve. Es sind jedoch auch andere Reihenfolgen möglich, was dann entsprechend andere Kurven ergibt. Ein wichtiger Spezialfall sind Peanokurven, in denen die Verfeinerungen *sortiert* auftreten, wo also alle Verfeinerungsschritte in einer Dimension direkt aufeinander folgen. Solche Kurven haben eine einfache, mäanderförmige Gestalt.

Die Einteilung in drei Teilbereiche bei der Peanokurve bringt entscheidende Vorteile: Die Ein- und Austrittspunkte der Kurve in jedem Bereich befinden sich in diagonal gegenüberliegenden Ecken. Bei der Zerlegung in drei Teile können alle Teilbereiche die selbe Struktur erhalten, wobei der mittlere Block gespiegelt wird. So bleiben die Ein- und Austrittspunkte an der selben Stelle und es ergibt sich wieder eine zusammenhängende Kurve (Abb. 4.2). Prinzipiell funktioniert dies mit jeder ungeraden Anzahl an Teilen. Praktisch tritt jedoch das Problem auf, dass eine gleichmäßig verfeinerte Peanokurve mit einer t-Teilung und n Verfeinerungen pro Dimension t^n Zellen in jeder Dimension besitzt. Für größere t wächst die Anzahl Zellen also sehr schnell und ist schwer an praktische Problemgrößen anzupassen; deshalb bietet es sich an das kleinstmögliche $t = 3$ zu wählen.

Mit diesem Ansatz lässt sich die Peanokurve auch problemlos auf beliebig viele Dimensionen verallgemeinern. Solche höherdimensionalen Peanokurven haben die Eigenschaft, dass jede Projektion auf eine Teilmenge der Dimensionen wieder eine Peanokurve darstellt. Diese Eigenschaft erlaubt es auch, viele Algorithmen, die auf Basis von Peanokurven arbeiten, für beliebige Dimensionszahlen zu verallgemeinern.

4.2 Grammatiken

Die rekursive Struktur von raumfüllenden Kurven präzise zu beschreiben ist nicht einfach. Ein nützliches Werkzeug stellen hier kontextfreie Grammatiken dar, wie sie üblicherweise zur rekursiven Beschreibung formaler Sprachen benutzt werden. Eine direkte Beschreibung einer Peanokurve mit Hilfe einer Grammatik zeigt Abb. 4.3. Solche grafischen Beschreibungen sind jedoch eher unpraktisch. Im Folgenden werden deshalb andere Arten von Grammatiken eingeführt, die raumfüllende Kurven beschreiben können ohne auf grafische Darstellungen zurückzugreifen. Die grundlegende Idee ist es, anstelle der Kurve selbst, *Befehlsworte* zu erzeugen, die die Konstruktion der Kurve beschreiben.

An der grafischen Grammatik werden jedoch schon einige Schwierigkeiten der Darstellung mit Grammatiken deutlich. Aus den rekursiven Regeln der Kurve allein lässt sich kein Terminalwort ableiten; dies kann am ehesten als eine Darstellung der unendlichen rekursiven Struktur der eigentlichen raumfüllenden Kurve verstanden werden. Diese ist für unsere Anwendung aber nicht von Bedeutung, weshalb Abbruchregeln eingeführt werden, die die Rekursion beenden und die Kurve als Terminalwort erzeugen. Im Sinne der formalen Sprachen wäre dann aber jede mögliche Ableitung eines Terminalworts, insbesondere auch ungleichmäßig verfeinerte Varianten, zulässig. Um solche Fälle auszuschließen, wird die zusätzliche Forderung gestellt, dass alle Nichtterminale in einer Produktionsregel auf einmal

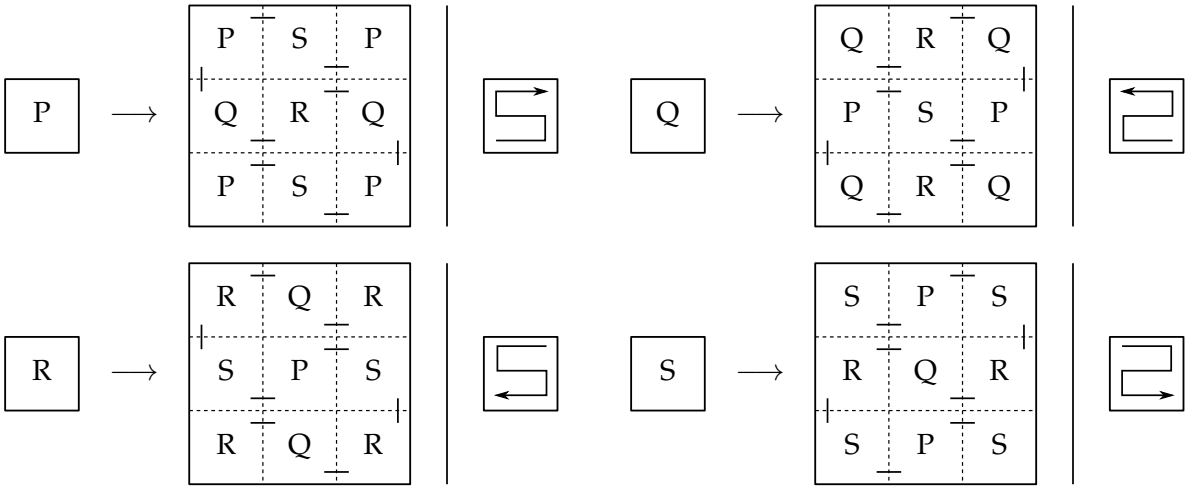

Abbildung 4.3: Eine grafische Grammatik mit den Nichtterminalen P, Q, R, S, die direkt Peanokurven beschreibt. Neben den vier rekursiven Regeln gibt es Abbruchregeln, die es erlauben, eine endlich verfeinerte Kurve zu erzeugen.

ersetzt werden müssen, und dass alle Nichtterminale auf einmal durch Terminale ersetzt werden müssen.

Eine andere Möglichkeit ist es, die Produktionsregeln zu vervielfachen und mit Indices zu versehen, so dass genau eine Ableitung eines Terminalworts möglich ist. Während solche Grammatiken eher unpraktikabel zur allgemeinen Beschreibung raumfüllender Kurven sind, entspricht dies am ehesten der algorithmischen Realisierung mit rekursiven Abstieg.

4.2.1 Typen

Für die algorithmische Umsetzung wird die Generierung der raumfüllenden Kurve auf zwei Komponenten verteilt: Einerseits eine Grammatik, die ein Befehlswort erzeugt, und andererseits ein *Operationswerk*, das die Befehle der Grammatik umsetzt. Je nach verwendetem Operationswerk, genauer der Art und Menge an Zustandsinformation im Operationswerk, unterscheidet sich die Form der benötigten Grammatik. In diesem Abschnitt werden drei Arten von Grammatiken eingeführt: Plottergrammatiken, Turtlegrammatiken und *hierarchische Grammatiken*. Die beiden ersteren sind aus der Literatur [Bad13] bekannt; letztere werden hier eingeführt, um auch die hierarchische Struktur der Peanokurve nutzen zu können, die für die Hierarchisierung nützlich ist.

Plottergrammatiken benutzen unter den hier eingeführten Typen das einfachste Operationswerk: das Modell ist hierbei ein einfacher Stift, der durch die Befehle der Grammatik um einen Schritt in einer Himmelsrichtung bewegt werden kann. Dafür ergeben sich relativ komplexe Grammatiken: für eine zweidimensionale Peanokurve mit gleichmäßiger, abwechselnder

Verfeinerung werden vier Nichtterminale benötigt, die jeweils eine gedrehte und gespiegelte Version des 3×3 Grundmusters darstellen. Es ergibt sich das Regelwerk [Bad13]

$$\begin{aligned} P &\longrightarrow P \uparrow Q \uparrow P \rightarrow S \downarrow R \downarrow S \rightarrow P \uparrow Q \uparrow P \\ Q &\longrightarrow Q \uparrow P \uparrow Q \leftarrow R \downarrow S \downarrow R \leftarrow Q \uparrow P \uparrow Q \\ R &\longrightarrow R \downarrow S \downarrow R \leftarrow Q \uparrow P \uparrow Q \leftarrow R \downarrow S \downarrow R \\ S &\longrightarrow S \downarrow R \downarrow S \rightarrow P \uparrow Q \uparrow P \rightarrow S \downarrow R \downarrow S \end{aligned} \tag{4.2}$$

mit den Terminalen $\uparrow, \downarrow, \leftarrow, \rightarrow$ für die entsprechenden Bewegungen des Stiftes. Prinzipiell kann jedes Nichtterminal als Startsymbol benutzt werden, da jedes eine Peanokurve erzeugt, per Konvention wählen wir P. Um ein Terminalwort für eine endliche Verfeinerung zu erzeugen, müssen nach der gewünschten Anzahl Verfeinerungen alle Nichtterminale gelöscht werden. Es wird angenommen, dass immer alle Nichtterminale auf einmal ersetzt werden, wie oben beschrieben.

Turtlegrammatiken erweitern das Operationswerk um eine zusätzliche Richtungsinformationen, um anstelle der absoluten Himmelsrichtungen relative Angaben wie *links* oder *rechts* benutzen zu können. Die Idee beruht auf der simulierten Schildkröte, die gerne in Programmiereinführungen benutzt wird und sich mit den Befehlen *links*, *rechts*, *vor* steuern lässt. Durch die zusätzliche Richtungsinformation können die Regeln für P und R bzw. Q und S, die jeweils gedrehte Varianten des selben Musters darstellen, zusammengefasst werden:

$$\begin{aligned} P &\longrightarrow P\ vor\ Q\ vor\ P\ rechts\ vor\ rechts\ Q\ vor\ P\ vor\ Q\ links\ vor\ links\ P\ vor\ Q\ vor\ P \\ Q &\longrightarrow Q\ vor\ P\ vor\ Q\ links\ vor\ links\ P\ vor\ Q\ vor\ P\ rechts\ vor\ rechts\ Q\ vor\ P\ vor\ Q \end{aligned} \tag{4.3}$$

Um die Grammatik noch weiter zu reduzieren und auch höherdimensionale Varianten zu ermöglichen, bieten sich *dimensionsrekursive* Grammatiken an, die je nur eine Verfeinerung in drei Blöcke betrachten und die Raumdimensionen rekursiv nacheinander abarbeiten. Für zwei Dimensionen ergeben sich folgende Regeln:

$$\begin{aligned} P &\longrightarrow links\ Q\ rechts\ vor\ rechts\ P\ links\ vor\ links\ Q\ rechts \\ Q &\longrightarrow rechts\ P\ links\ vor\ links\ Q\ rechts\ vor\ rechts\ P\ links \end{aligned} \tag{4.4}$$

Um diese nun für höhere Dimensionszahlen zu verallgemeinern, müssten die Begriffe *links* und *rechts* verallgemeinert werden. Dies ist prinzipiell möglich, allerdings sind solche Grammatiken weniger anschaulich. Deshalb werden im Folgenden für mehr als zwei Dimensionen nur Plottergrammatiken benutzt.

Für die Hierarchisierung erweitern wir das Operationswerk noch weiter, indem die Schildkröte oder auch der Plotter mit Informationen über die Größe bzw. die Verfeinerung der aktuell betrachteten Zelle versehen wird. Das Ziel der auf raumfüllenden Kurven basierenden Algorithmen ist es ja meist nicht, die Kurve zu zeichnen, sondern beim Durchlaufen der Kurve verschiedene Operationen auszuführen, wobei die Informationen zur aktuellen Zelle hilfreich sind. Die Befehle lauten dann *auf* und *ab*, um hierarchisch auf- und abzusteigen, *vor*, um in die nächste benachbarte Zelle der aktuellen Verfeinerung zu wechseln, sowie *links* und *rechts* um bei Turtlegrammatiken die aktuelle Blickrichtung zu verändern. Bei

Plottergrammatiken werden stattdessen die Operationen um einen Index erweitert, der die Dimension angibt, in der eine Veränderung stattfinden soll, sowie um eine Richtungsangabe (aufsteigende/absteigende Koordinatenwerte). Die *ab*-Operation wechselt dabei immer in die erste Zelle der angegebenen Richtung, so dass mit zwei *vor*-Schritten die entsprechende Verfeinerung abgearbeitet werden kann. Später wird zudem verlangt, dass *auf* nur in der letzten Zelle und *vor* nur genau zweimal in jedem Verfeinerungsschritt ausgeführt werden, was für die Peanokurve sowieso automatisch der Fall ist.

Die Grammatiken müssen nun so angepasst werden, dass sie die zusätzlichen Befehle erzeugen. Für die dimensionsrekursiven Turtlegrammatiken ist das einfach möglich, da diese genau die selbe hierarchische Struktur benutzen:

$$\begin{array}{lll} P & \longrightarrow & ab\ rechts\ Q\ links\ vor\ links\ P\ rechts\ vor\ rechts\ Q\ links\ auf \\ Q & \longrightarrow & ab\ links\ P\ rechts\ vor\ rechts\ Q\ links\ vor\ links\ P\ rechts\ auf \end{array} \tag{4.5}$$

Plottergrammatiken sind etwas aufwändiger, da die Richtungsinformation in der Grammatik mitgeführt werden muss. Durch Parametrisierung der Regeln können solche Grammatiken aber ebenfalls sehr kompakt dargestellt werden:

$$\begin{array}{lllllllll} P_i & \longrightarrow & ab_{\,i,1} & Q_{i+1} & vor_{\,i,1} & P_{i+1} & vor_{\,i,1} & Q_{i+1} & auf_{\,i,1} \\ Q_i & \longrightarrow & ab_{\,i,-1} & P_{i+1} & vor_{\,i,-1} & Q_{i+1} & vor_{\,i,-1} & P_{i+1} & auf_{\,i,-1} \end{array} \tag{4.6}$$

Die Indices i für die Dimension sind dabei modulo d zu interpretieren, der zweite Index gibt jeweils die Richtung an: 1 für in Koordinatenrichtung, -1 für entgegen der Koordinatenrichtung. Offensichtlich erzeugt diese Grammatik direkt eine d-dimensionale Kurve. Die dafür nötige, variable Anzahl Regeln liegt versteckt in den Indices. Durch Mitführen der Richtung als weiteren Parameter kann die Grammatik noch kompakter dargestellt werden, allerdings auf Kosten der Lesbarkeit:

$$P_{i,r} \longrightarrow ab_{\,i,r} P_{i+1,-r}\, vor_{\,i,r} P_{i+1,r}\, vor_{\,i,r} P_{i+1,-r}\, auf_{\,i,r}\,. \tag{4.7}$$

Für die Implementierung wird diese Form bevorzugt, da sie duplizierten Code für die Regeln P und Q vermeidet. Ein weiterer Vorteil ist, dass sich durch Interpretation des Parameters i als Anzahl Verfeinerungsschritte auch nicht-abwechselnde Verfeinerungen realisieren lassen.

4.3 Datentransport

Der Hauptnutzen der Peanokurve für die in dieser Arbeit vorgestellten Algorithmen ist der, dass es möglich ist, entlang der Peanokurve die Zellen eines regulären Gitters zu durchlaufen und in jeder Zelle die Eckknoten zu betrachten, wobei alle Werte im Sinne eines Stack & Stream Algorithmus über Stacks transportiert werden. Die Schwierigkeit liegt dabei darin, dass jeder Knoten in mehreren Zellen benötigt wird, die bei einer Traversierung nicht direkt nacheinander durchlaufen werden können. Es müssen also Daten zwischengespeichert werden. Im Folgenden geschieht das auf Stacks, was durch den mäanderförmigen Verlauf

der Peanokurve ermöglicht wird: Die Peanokurve durchläuft aneinander grenzende Zeilen oder Spalten in entgegengesetzter Richtung, was zum last-in-first-out-Prinzip eines Stacks passt.

In diesem Abschnitt werden zwei Mechanismen für den Datentransport vorgestellt, die in der Regel zusammen benutzt werden, aber weitgehend unabhängig voneinander arbeiten.

4.3.1 Knotenstacks

Die *Knotenstacks* ermöglichen den Datentransport beim hierarchischen Ab- und Aufstieg in einer Dimension. Über die Knotenstacks werden auch die Daten transportiert, die im zeitlichen Ablauf direkt nacheinander benötigt werden.

Für ein d-dimensionales Gitter gibt es 2^d Knotenstacks, die mit den Ecken der jeweils betrachteten Zelle assoziiert werden. Auf dieses Konstrukt lassen sich nun direkt die Operationen einer hierarchischen Grammatik abbilden:

- *ab*: Auf die in Verfeinerungsrichtung vorn liegenden Stacks die neuen Knoten der verfeinerten Zelle legen.
- *auf*: Die in Verfeinerungsrichtung hinteren Knoten von den Stacks nehmen; darunter liegen die Knoten der hierarchisch übergeordneten Zelle.
- *vor*: Hier müssen das erste und zweite *vor* unterschieden werden: beim ersten werden die noch verbleibenden, hinteren Knoten der übergeordneten Zelle mit neuen Knoten verdeckt, beim zweiten werden die nicht mehr benötigten Knoten der vorherigen Zelle abgehoben um die Knoten der hierarchisch übergeordneten Zelle wieder sichtbar zu machen.

Nach jeder dieser Operationen liegen die Eckknoten der aktuellen Zelle oben auf den Knotenstacks. Allerdings liegt nicht immer die (in Koordinatenrichtung) selbe Ecke auf dem selben Stack. Nach der ersten *vor*-Operation erscheinen die Ecken gespiegelt, nach der zweiten wieder wie zuvor.

Um die Steuerung der Knotenstacks zu vereinfachen, werden für die Implementierung die Knotenstacks selbst zum Operationswerk: die Grammatik erzeugt dann anstelle der symbolischen Befehle *auf*, *ab* und *vor* direkt die *push*- und *pop*-Befehle für die Knotenstacks. Zudem ist es sinnvoll, nach den *vor*-Operationen durch Umbenennen der Stacks die Spiegelung aufzuheben. Dazu wird eine neue Operation eingeführt. Die Implementierung nutzt folgende Bezeichnungen:

- PUSHFRONT(dim, dir): neue Werte vom Input auf die in Dimension dim und Richtung dir vorderen Stacks legen.
- POPBACK(dim, dir): Werte von den hinteren Stacks abheben und auf den Output legen.
- FLIP(dim): die Knotenstacks umbenennen, um eine Spiegelung an der Ebene orthogonal zu Dimension dim zu erreichen.

Algorithmus 4.1 Traversierung einer Peanokurve mit l eindimensionalen Verfeinerungen mit Hilfe der Knotenstacks.

```
procedure P(l, dir = 1)
    if l = 0 then
                                        // Hier Sonderbehandlung auf Gitterzellen ausführen
    else
                                // Hier Operationen auf hierarchischen Zellen im Abstieg ausführen
        dim ← l mod d
        PUSHFRONT(dim, dir)
        P(l − 1, −dir)
        FLIP(dim)
        PUSHFRONT(dim, dir)
        P(l − 1, dir)
        POPBACK(dim, dir)
        FLIP(dim)
        P(l − 1, −dir)
        POPBACK(dim, dir)
                                // Hier Operationen auf hierarchischen Zellen im Aufstieg ausführen
    end if
end procedure
```

Algorithmus 4.1 realisiert eine Grammatik ähnlich der in (4.7) mittels rekursivem Abstieg.

4.3.2 Transportstacks

Der im letzten Abschnitt eingeführte Algorithmus hat noch immer den Nachteil, dass die meisten Werte mehrere Male vom Input gelesen und auf den Output geschrieben werden. Abb. 4.4 zeigt das Problem an einem sehr einfachen Beispiel: Während die Knotenstacks für den Transport zur direkt folgenden Zelle genügen, werden später in der Traversierung Knoten wieder benötigt, die nicht mehr auf den Knotenstacks liegen.

Um dies zu verhindern, wird nun die Eigenschaft der Peanokurve, dass benachbarte, aber nicht direkt nacheinander durchlaufene Zellen in einer LIFO-Reihenfolge besucht werden, genutzt.

Am Peano-Grundmuster sind zwei Dinge erkennbar, hier zur Erläuterung der zweidimensionale Fall: Zunächst besteht das Grundmuster aus 3×3 Zellen, für die jedoch 4×4 Knoten betrachtet werden müssen. Die Knoten am Rand teilt das betrachtete Muster mit den benachbarten. Schlägt man diese Knoten je nur einem Muster zu, so bleiben nur 3×3 Knoten, bis auf Ausnahmen am Rand. Für die Knoten im Inneren des Musters wird die LIFO-Eigenschaft durch das Grundmuster selbst erfüllt. Für die Knoten am Rand ist die LIFO-Eigenschaft ebenso erfüllt, wenn man das hierarchisch übergeordnete Muster betrachtet (Abb. 4.5). Eine Ausnahme bilden wieder die Knoten am Rand, diese stellen aber ohnehin kein Problem dar, da sie nur einmal benötigt werden.

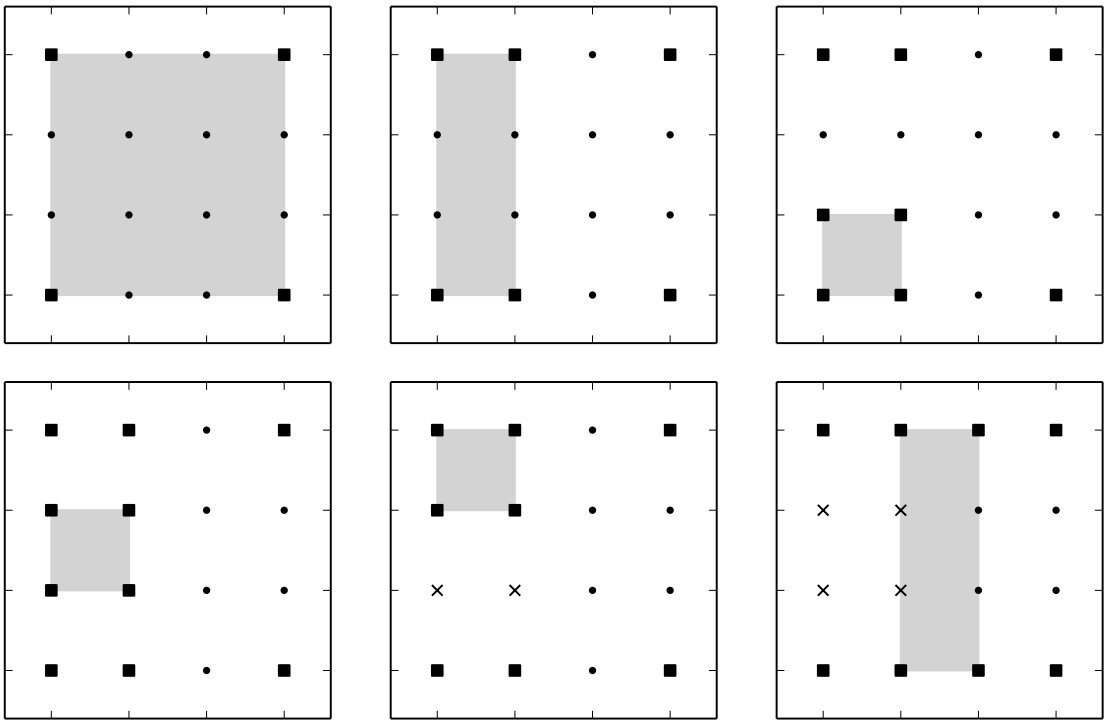

Abbildung 4.4: Beginn einer Traversierung mit einer 1- und einer 2-Verfeinerung ohne Transportstacks. Die Symbole geben den Stack an, auf dem ein Knoten liegt: □ Knotenstacks, ○ Input, × Output, die aktuell betrachtete hierarchische Zelle ist grau gefärbt. Im folgenden Schritt müsste ein Knoten nochmal gelesen werden, der schon auf dem Output liegt.

Wir benutzen nun 2*d* *Transportstacks*, die mit den Raumdimensionen assoziiert werden, wobei jeweils ein Stack gelesen, der andere geschrieben wird. An jeder Seite könnten andere Zellen benachbart sein, zu denen oder von denen Daten transportiert werden müssen. Die grundlegende Idee ist es nun, die Knoten, die in der nächsten Zelle noch benötigt werden, statt auf den Output, auf den der entsprechenden Richtung zugeordneten Stack zu legen, und umgekehrt, die Knoten, die mit der vorherigen Zelle geteilt werden, von der entsprechenden Seite statt des Inputs zu lesen. Da dies auf jeder Ebene des hierarchischen Abstiegs geschieht, muss stets mehr als eine benachbarte Zelle betrachtet werden; die Verfeinerungen in einer Dimension können sich jedoch den selben Stack teilen. Die Eigenschaften der Peanokurve garantieren, dass die Knoten in der Reihenfolge auf den Transportstacks zum liegen kommen, in der sie benötigt werden.

Zur Erklärung betrachten wir zunächst einen Fall *ohne* Knotenstacks, es erfolgt also der gesamte Datentransport über Transportstacks. Wir betrachten noch immer jeweils eine Zelle im hierarchischen Abstieg, an deren Ecken sich nun je genau ein Knoten befinden kann. Ein *vor*-Schritt besteht jetzt aus zwei Phasen: zunächst wird die vorherige Zelle verlassen, d. h. alle Knoten auf den Ecken der Zelle werden auf Transportstacks bzw. den Output gespeichert.

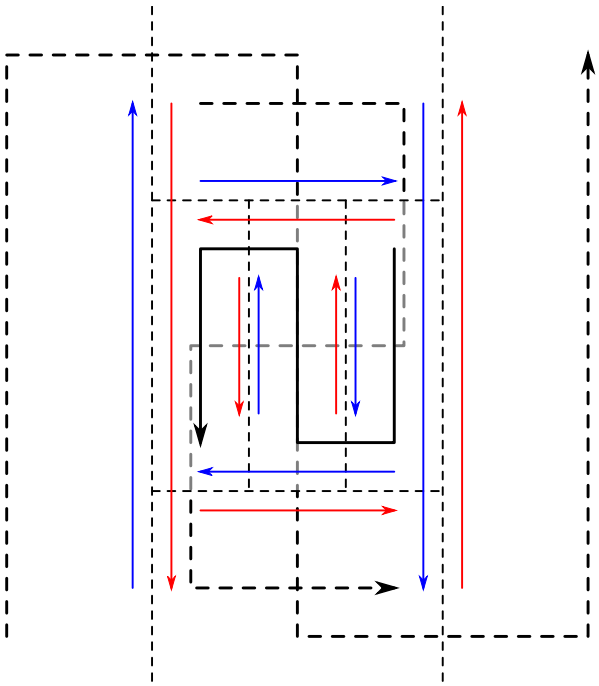

Abbildung 4.5: Ein Grundmuster (mitte) und die Abarbeitungsreihenfolge der angrenzenden Knoten: Im Inneren gewährleistet das Grundmuster, dass die Knoten beim zweiten Besuch (rot) umgekehrt zum ersten (blau) gelesen werden, an den Rändern die übergeordneten Grundmuster.

Dann wird die nächste Zelle betreten, dazu werden auf die Ecken wieder Knoten von den Transportstacks und dem Input gelegt. Abb. 4.6 zeigt, auf welche Transportstacks $t_1, \ldots, t_d$ Knoten gelegt werden bzw. von welchen Stacks $t'_1, \ldots, t'_d$ Knoten gelesen werden. Damit überhaupt Daten transportiert werden können, muss der Zusammenhang von t_k und t'_k noch festgelegt werden: nach jedem Schritt in Dimension k werden t_k und t'_k vertauscht, damit die bereits eingelesenen aber noch einmal benötigten Werte zur Verfügung stehen.

Nun muss betrachtet werden, *wie schnell* sich die Kurve in den jeweiligen Dimensionen bewegt. Im gezeigten Fall wird ein Schritt nach oben ausgeführt, die vertikale Bewegung ist also am schnellsten. Die Hälfte der Knoten (4 Stück) wird über diesen Stack transportiert. Die horizontale Bewegung nach links erfolgt auf einer hierarchisch gröberen Stufe, es finden also weniger Schritte in horizontaler Richtung statt, die Bewegung ist in diesem Sinne *langsamer*. Über die Transportstacks in dieser Dimension werden je nur 2 Knoten, also die Hälfte der verbleibenden Knoten, transportiert. Dafür müssen diese Knoten länger auf dem Transportstack verbleiben, da sie erst nach mehreren Schritten in vertikaler Richtung wieder benötigt werden. Am langsamsten ist die Bewegung in z-Richtung (Dimension 3), hier wird je nur ein Knoten transportiert. Es verbleibt je noch ein Knoten der vom Input gelesen oder auf den Output geschrieben wird.

Nach dem angedeuteten Schema lässt sich für beliebig hochdimensionale Räume angeben, welche Knoten über welche Stacks transportiert werden müssen, damit jeder Knoten nur

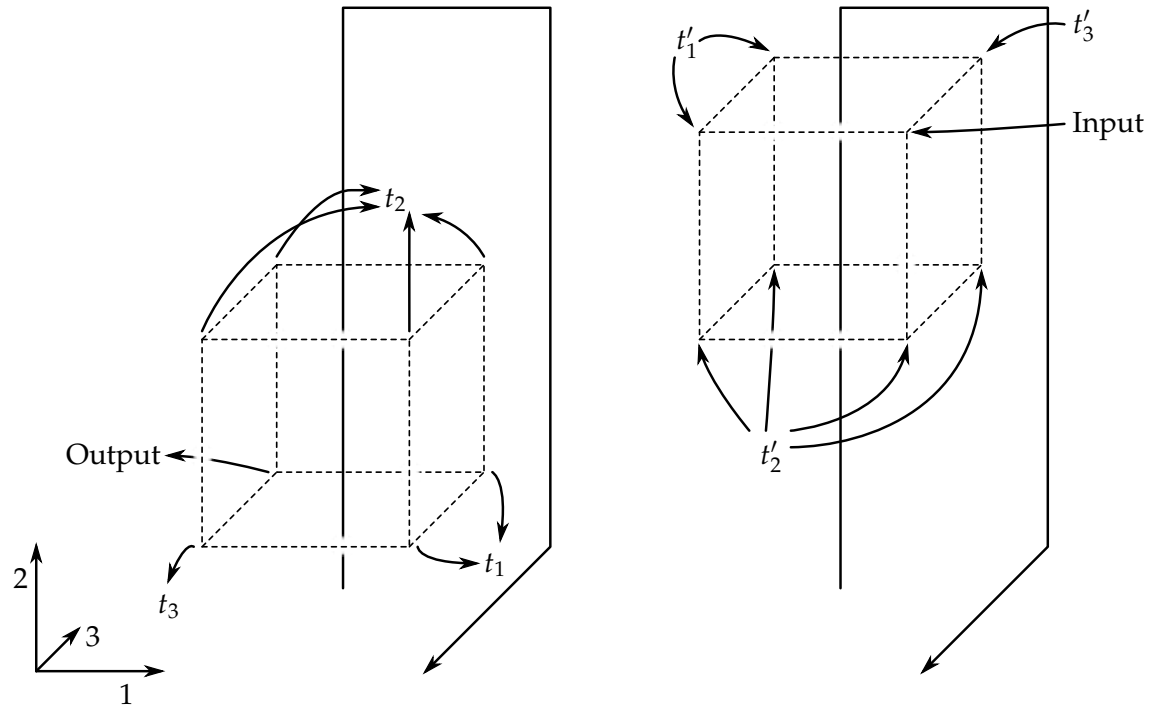

Abbildung 4.6: Verlassen und Betreten einer Zelle bei einer dreidimensionalen Peanokurve ohne Knotenstacks. Die Peanokurve verläuft zunächst von unten nach oben, dann von links nach rechts, und schließlich von hinten nach vorn.

einmal vom Input eingelesen werden braucht. Für eine praktische Realisierung ist zudem noch eine Ausnahmebehandlung für die Ränder nötig, zudem müssen die hierarchischen Schritte *auf* und *ab* realisiert werden. Hier können einfach die Knotenstacks zur Hilfe genommen werden: die Knotenstacks übernehmen den Datentransport zwischen den direkt aufeinander folgenden Zellen sowie im hierarchischen Abstieg, die Transportstacks erledigen den verbleibenden Datentransport.

Ein Problem ist allerdings, dass die bisher eingeführten Grammatiken nicht genügend Informationen mitführen, um die Richtungen und Geschwindigkeiten für jede Dimension zu kennen. Eine Option ist es, diese Informationen im Operationswerk mitzuführen. Aus Implementierungssicht ist es jedoch wünschenswert, wenig Zustandsinformation im Operationswerk zu haben, und diese Informationen stattdessen in der Grammatik mitzuführen.

Im Hinblick auf die spätere Verwendung betrachten wir direkt einen d-dimensionalen Raum und eine möglicherweise ungleichmäßige Verfeinerung. Die Grammatik enthält also Regeln, die es erlauben, aus jeder d-dimensionalen Zelle mit bekannten Bewegungsrichtungen und -geschwindigkeiten in einer beliebigen Dimension k zu verfeinern. Aus dieser Grammatik lassen sich alle später verwendeten Peanokurven ableiten, und es ist möglich, den gesamten Datentransport über Knoten- und Transportstacks zu lösen, sodass jeder Knoten nur einmal vom Input gelesen und nur einmal auf den Output geschrieben wird. Die mitgeführten

Informationen reichen zunächst jedoch nur für sortierte Verfeinerungen, für allgemeine Peanokurven müssen noch mehr Daten mitgeführt werden.

Die große Anzahl Regeln wird wieder durch eine parametrisierte Regel beschrieben:

$$\begin{array}{lllll} P_{\mathbf{p},\,\mathbf{r},\,\mathbf{w}} \longrightarrow & ab_{\,k,\,p_k} & P_{\mathbf{p},\,\mathbf{r}\setminus k\circ k,\,\mathbf{w}|_k:=p_k} & & \\ & vor_{\,k,\,p_k} & P_{\mathbf{p}\oplus\mathbf{1}-\mathbf{e}_k,\,\mathbf{r}\setminus k\circ k,\,\mathbf{w}|_k:=p_k} & & \\ & vor_{\,k,\,p_k} & P_{\mathbf{p},\,\mathbf{r}\setminus k\circ k,\,\mathbf{w}|_k:=p_k} & auf_{\,k,\,dp_k} & \mid \varepsilon \end{array} \tag{4.8}$$

Dabei bezeichnen

- $\mathbf{p}$ und $\mathbf{w}$ Multiindices mit je d Einträgen, die die Bewegungsrichtung in der jeweiligen Dimension angeben. Mögliche Werte sind 1 (in Koordinatenrichtung) und 0 (entgegen der Koordinatenrichtung).
- $\mathbf{r}$ einen Multiindex, der die Reihenfolge der Dimensionen angibt. Der letzte Eintrag gibt die Dimension an, in der die Bewegung am schnellsten ist, der erste die langsamste. Dieser Multiindex ist also immer eine Permutation von $\{1, \ldots, d\}$.
- $\mathbf{r} \setminus k$ den Multiindex $\mathbf{r}$, aus dem der Eintrag mit Wert k entfernt wurde, und die verbleibenden Einträge zusammengerückt.
- $\mathbf{r} \circ k$ den Multiindex $\mathbf{r}$, an dessen Ende k angehängt wurde.
- $\mathbf{p} \oplus \mathbf{x}$ die elementweise exklusiv-Oder Operation bzw. Addition in $\mathbb{Z}/2\mathbb{Z}$ auf Multiindices.
- $\mathbf{1} - \mathbf{e}_k$ einen Multiindex mit 0 an Stelle k, sonst 1.
- $\mathbf{w}|_k := x$ den Multiindex $\mathbf{w}$, dessen Eintrag k auf x gesetzt wurde.
- ε das leere Wort, also den Rekursionsabbruch.

Alle anderen Symbole sind wie zuvor eingeführt. Die Indices erfüllen zwei Aufgaben: $\mathbf{p}$ (Plan) gibt an, wie die aktuelle Zelle in den nächsten Verfeinerungen durchlaufen wird. Dies kann als Diagonale durch die Zelle aufgefasst werden, die den Eintritts- mit dem Austrittspunkt verbindet. Die Indices $\mathbf{w}$ (Weg) und $\mathbf{r}$ (Reihenfolge) beschreiben dagegen wie die Zelle erreicht wurde: $\mathbf{r}$ gibt die Dimensionen und $\mathbf{w}$ die zugehörigen Richtungen der bisherigen Verfeinerungen an, die für die Transportstacks bekannt sein müssen. Abb. 4.7 stellt einige der Regeln für drei Dimensionen grafisch dar.

Mit Hilfe der Indices $\mathbf{p}$ und $\mathbf{r}$ können nun die Transportstacks so wie oben beschrieben gesteuert werden, so dass *vor*, *auf* und *ab* die Knoten von den richtigen Stacks lesen. Analog zu Alg. 4.1 können die Operationen auch für die Knotenstacks formuliert und eine recursive-Descent Implementierung realisiert werden.

Zudem ist noch eine Sonderbehandlung der Zellen am Rand nötig. Eine Möglichkeit ist es, die Knoten am Rand zu Beginn auf den Transportstacks zu lagern, und das Transportschema unverändert durchzuführen. Nach der Traversierung befinden sich die Knoten des gegenüberliegenden Randes auf den Transportstacks, so dass die nächste Traversierung in

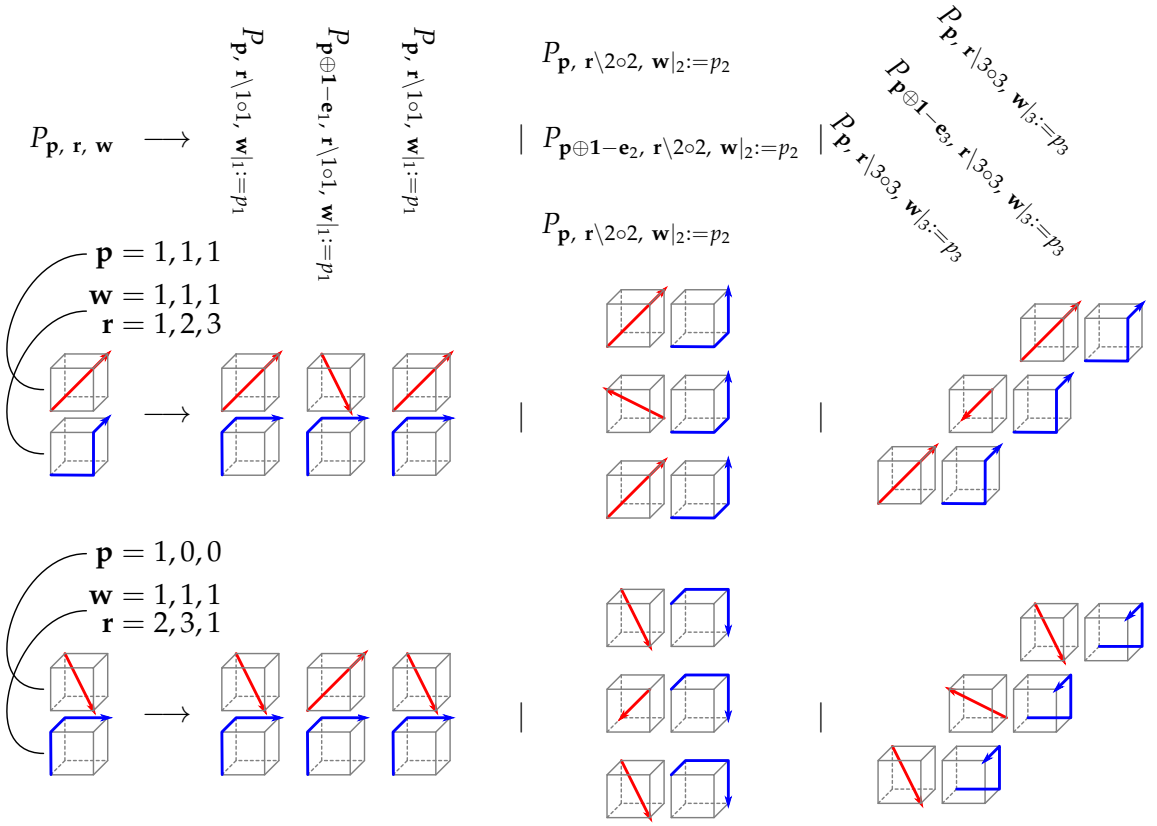

Abbildung 4.7: Grafische Veranschaulichung einiger Regeln aus Grammatik 4.8 für drei Dimensionen. Für zwei Werte von $\mathbf{p}, \mathbf{r}, \mathbf{w}$ werden die Regeln für die drei möglichen Werte von k dargestellt. Der Weg durchläuft die Dimensionen in der Reihenfolge von $\mathbf{r}$ mit den Richtungen aus $\mathbf{w}$. Die Diagonale $\mathbf{p}$ bestimmt die weiteren Verfeinerungen, die $\oplus$-Operation entspricht einer Spiegelung der Diagonale in allen Dimensionen außer der Verfeinerungsrichtung.

Gegenrichtung direkt erfolgen kann. Dies wird später als *getrennte* Speicherung bezeichnet. Die Randknoten werden getrennt von den übrigen Knoten gelesen/geschrieben, und es wird für jede Zelle genau ein Knoten vom Input gelesen. Diese Lösung kommt ohne weitere Informationen aus, ist jedoch nur für sortierte Verfeinerungen geeignet.

Für viele Situationen ist es günstiger, die Randknoten von Input zu lesen und auch wieder auf den Output zu schreiben, da dann das gesamte Gitter auf einem Stack gespeichert werden kann. Diese Darstellung wird als *flache* Speicherung bezeichnet. Für eine solche Sonderbehandlung wird die Grammatik um weitere Parameter erweitert, die für jede Dimension und Richtung angeben, ob die aktuelle Zelle am Anfang oder Ende des Wertebereichs dieser Dimension liegt. Da pro Zelle mehrere Knoten von Input gelesen werden müssen, und nicht eindeutig festgelegt ist, in welcher Reihenfolge dies geschieht, ist diese Form nicht eindeutig und in einer Implementierung müssen eventuell weitere Sonderfälle betrachtet werden. Die hier benutzte Implementierung erlaubt es, das Randverhalten für Input und Output

unabhängig zu wählen, was für einige spätere Algorithmen benötigt wird. Für abwechselnd verfeinerte Kurven ist diese Form unumgänglich, da eine Randbehandlung auch an den Zellgrenzen im Inneren des Bereiches notwendig ist. Die zusätzlichen Informationen können hier als Marker für jede Seite der Zelle aufgefasst werden, die angeben, ob die Freiheitsgrade an dieser Seite *noch nie* benötigt wurden, und ob sie *nie wieder* benötigt werden. Entsprechend wird im ersten Fall statt vom Transportstack vom Input gelesen, im zweiten Fall auf den Output statt des Transportstacks geschrieben. Bei abwechselnder Verfeinerung können auch beide Fälle zugleich auftreten, was die getrennte Speicherung wie oben beschrieben unmöglich macht; ein ähnliches Verhalten ließe sich mit einem zweiten Satz Transportstacks oder weiteren Inputstacks erreichen.

Der Datentransport ist an einem Beispiel in Abb. 4.8 dargestellt. Es wird ein Gitter mit zwei Verfeinerungen in jeder Dimension traversiert. Die Verfeinerungen in den beiden Dimensionen finden allerdings nicht abwechselnd statt, sondern, nach der Sprechweise aus Abschnitt 6.1.1 auf Seite 48, in der Anordnung 1221. Die vier Transportstacks werden durch in verschiedenen Richtungen orientierte Dreiecke dargestellt. Durch die Umbenennung der Transportstacks während der Traversierung kann sich das Symbol für einen Knoten auch ändern, ohne das dieser von einem Stack abgehoben oder darauf abgelegt wird. Im dargestellten Ausschnitt taucht keine Randbehandlung auf, die Randknoten werden getrennt gespeichert. Abb. 4.9 zeigt den Anfangszustand vor der Traversierung für flache und für die getrennte Speicherung. Für die in diesem Beispiel gewählte Traversierung ist eine getrennte Speicherung ohne zusätzliche Stacks möglich, obwohl es sich nicht um eine strikt sortierte Verfeinerung handelt. Solche Verfeinerungen werden in den später eingeführten Algorithmen oft verwendet. In Abb. 4.10 ist der Beginn der selben Traversierung zu sehen: Die Traversierung beginnt nicht auf Ebene der Gitterzellen, sondern steigt durch rekursive Verfeinerung dorthin ab. Die ersten Schritte sind der Traversierung sind identisch zu der in Abb. 4.4 auf Seite 31, da das hier gewählte Beispiel eine direkte Verfeinerung des vorherigen ist.

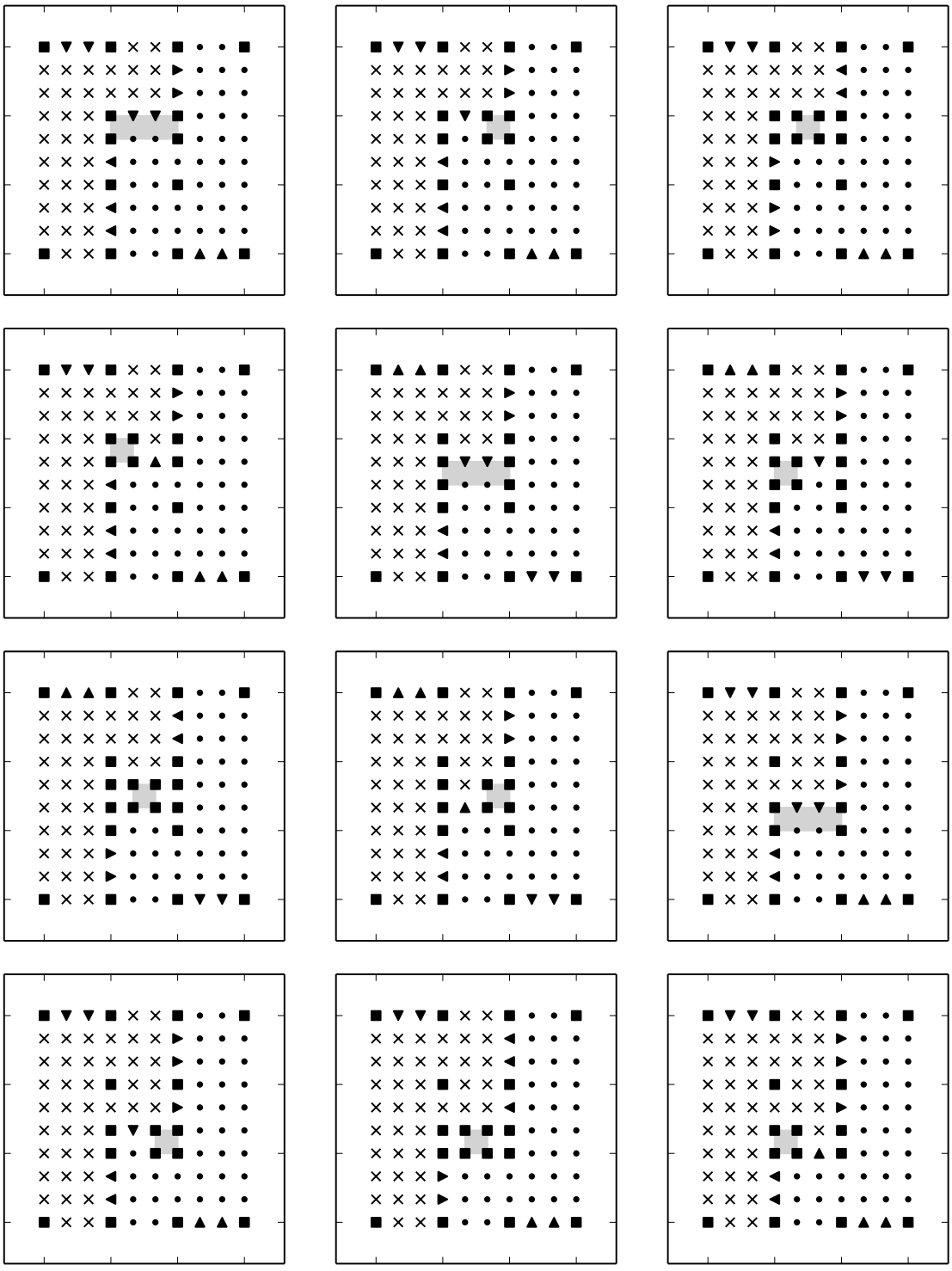

Abbildung 4.8: Traversierung mit Transportstacks (▷), Darstellung analog zu Abb. 4.4 auf Seite 31: □ Knotenstacks, ○ Input, × Output, die aktuell betrachtete hierarchische Zelle ist grau gefärbt.

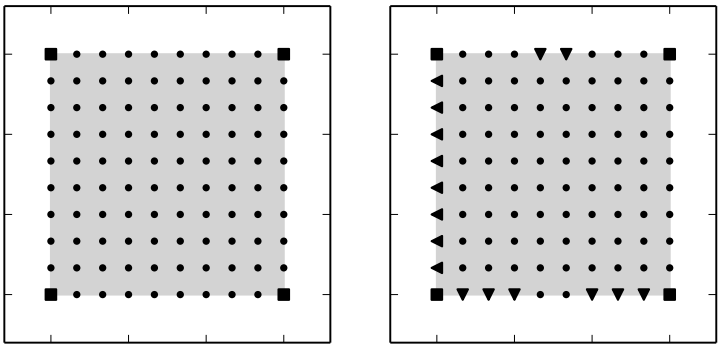

Abbildung 4.9: Anfangszustand für flache Speicherung (links) und getrennte Speicherung (rechts). Darstellung wie in Abb. 4.4 auf Seite 31: □ Knotenstacks, ∘ Input, × Output, ▷ Transportstacks.

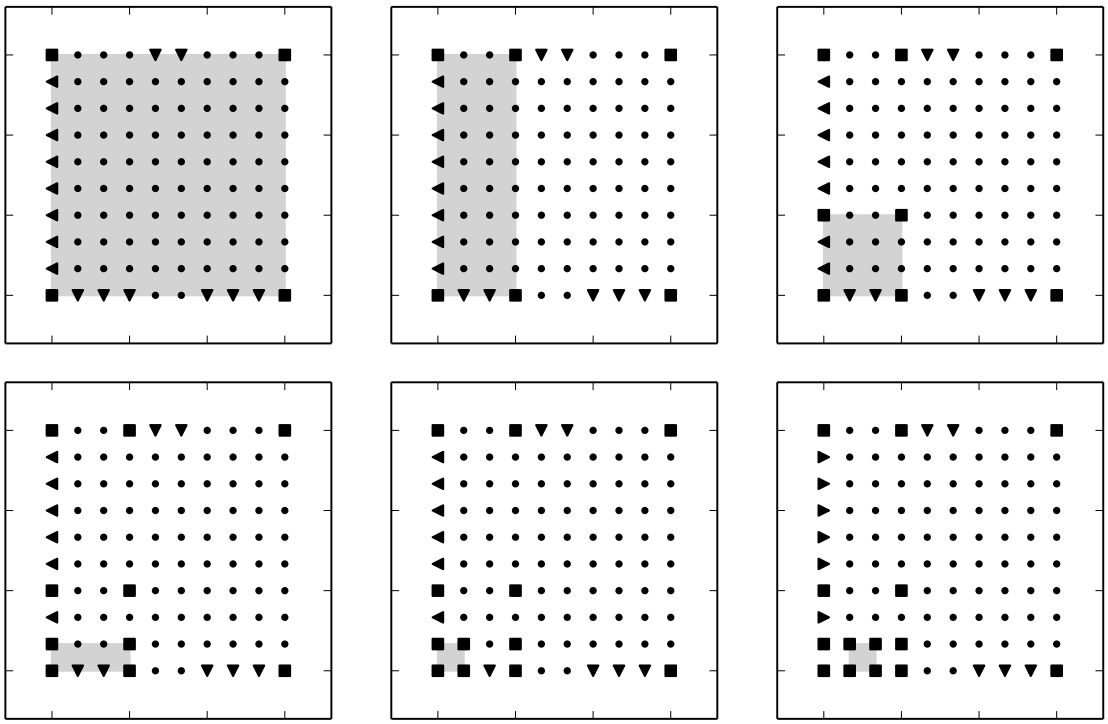

Abbildung 4.10: Die ersten Schritte der Traversierung: erkennbar ist der hierarchische Abstieg bis zu den Gitterzellen. Darstellung wie in Abb. 4.4: □ Knotenstacks, ∘ Input, × Output, ▷ Transportstacks.

5 Hierarchisierungsalgorithmen für volle Gitter

Wie in Kapitel 2 eingeführt wurde, handelt es ich bei der Hierarchisierung um eine Basistransformation aus der Knotenbasis in die hierarchische Basis, zumindest auf vollen Gittern. In diesem Kapitel werden Algorithmen für die Hierarchisierung auf vollen Gittern eingeführt, die auf verschiedenen Konzepten basieren. Die Cache-Effizienz steht dabei nicht im Mittelpunkt: bei vollen Gittern erfolgt der Großteil der Rechenoperationen auf den hierarchisch feinsten Ebenen, die, bei einer einfachen zeilenweisen Darstellung des Gitters im Speicher, meist eine starke räumliche Lokalität aufweisen. Die Cache-Effizienz von Stack & Stream-Algorithmen ist erst für dünne Gitter attraktiv, wo nur noch die gröberen Ebenen verbleiben und zudem bei den oft benutzten Hashmaps jegliche räumliche Lokalität verloren geht.

Alle hier vorgestellten Algorithmen laufen in Linearzeit, wenn man von konstanten Kosten für einen Speicherzugriff ausgeht. Für manche Elemente werden logarithmisch viele Operationen ausgeführt, der Aufwand dafür amortisiert sich aber durch die geringe Häufigkeit zu linearem Overhead. Dies wird in Abb. 5.1 durch eine umgekehrte Betrachtung deutlich: Für jeden Knoten müssen, unabhängig von der Teilung und der Verfeinerung, genau zwei Elternknoten für die Hierarchisierung betrachtet werden (Sonderfälle am Rand). Daraus ergeben sich direkt $\mathcal{O}(N)$ Operationen für die gesamte Hierarchisierung.

Die hier vorgestellten Algorithmen lassen sich ohne größere Probleme für Gitter mit beliebiger Teilung anpassen. Um konsistent zum folgenden Kapitel zu bleiben, werden sie hier für dreigeteilte Gitter beschrieben. Für den praktischen Einsatz sind die Algorithmen auf zweigeteilten Gittern meist günstiger: Die Anzahl Gitterzellen in jeder Dimension m muss bei einer t-Teilung die Form $t^k + 1$ haben, was für $t = 3$ schnell unpraktikabel groß wird. Zunächst werden nur eindimensionale Gitter betrachtet, mit Hilfe des *unidirektionalen Prinzips* können diese dann auf höhere Dimensionszahlen verallgemeinert werden.

5.1 Naiv-Rekursiv

Der grundlegende Hierarchisierungsalgorithmus, der hier als *naive Hierarchisierung* bezeichnet wird, basiert auf einer einfachen Rekursion. Diese folgt der rekursiven Verfeinerung des hierarchisierten Gitters und führt, im rekursiven Aufstieg, die Hierarchisierung aus. Auf die dazu benötigten Werte wird direkt zugegriffen, ohne Rücksicht auf die Cache-Effizienz.

Algorithmus 5.1 implementiert dieses Verfahren. Da die Hierarchisierung im Aufstieg erfolgt, ist sichergestellt, dass es keine Stellen mehr gibt, an denen der originale Funktionswert eines

Algorithmus 5.1 Der naive, rekursive Hierarchisierungsalgorithmus. Die Ein- und Ausgabe erfolgt im Array *data*, die Indices s und e markieren immer den Anfang und Ende des betrachteten Intervalls, das dann rekursiv verfeinert wird.

procedure NAIV-HIER($data$, $s = 1$, $e = m$)
  $k \leftarrow (e - s)/3$
  if $k > 1$ **then**
    NAIV-HIER($data$, s, $s + k$)
    NAIV-HIER($data$, $s + k$, $e - k$)
    NAIV-HIER($data$, $e - k$, e)
  end if
  $data[s + k] \leftarrow data[s + k] - \frac{2}{3} \cdot data[s] + \frac{1}{3} \cdot data[e]$
  $data[e - k] \leftarrow data[e - k] - \frac{1}{3} \cdot data[s] + \frac{2}{3} \cdot data[e]$
end procedure

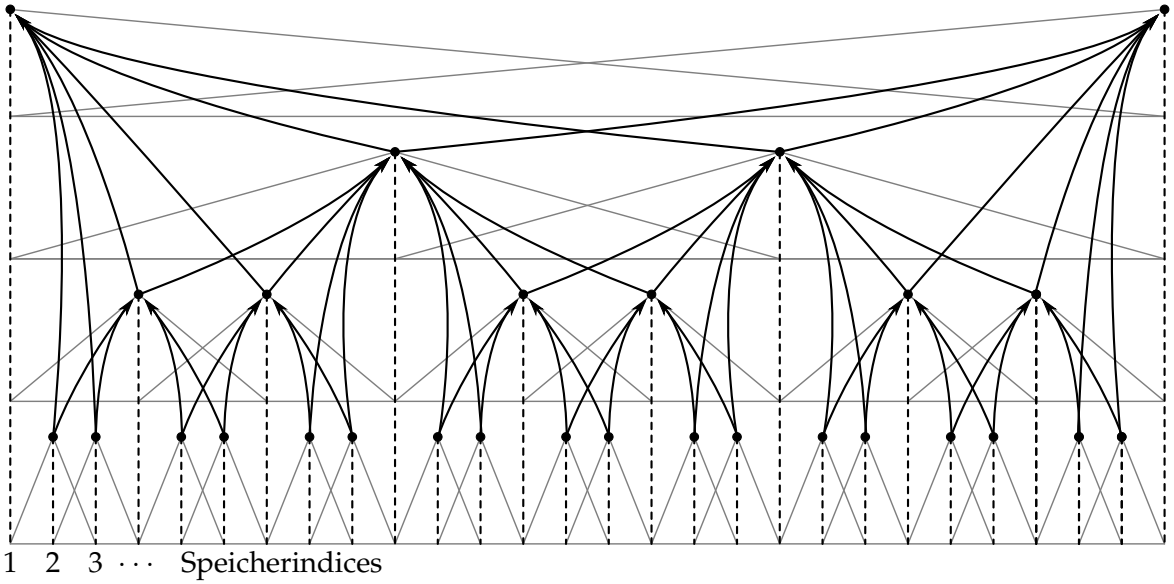

Abbildung 5.1: Daten-Abhängigkeitsgraph für die Hierarchisierung auf dreigeteilten Gittern. Alle Knoten haben Ausgangsgrad zwei, also zwei *Eltern* (Unabhängig von der Teilung) und Eingangsgrad $2(l_{max} - l)$ (für die Dreiteilung), also so viele *Kinder*, die den Funktionswert benötigen.

Knoten benötigt wird. Die Hierarchisierung darf also den originalen Wert überschreiben, und kann so in-place durchgeführt werden.

Bei diesem Algorithmus werden viele Werte mehrfach gelesen: die Knoten groben Verfeinerungslevels sind *hierarchische Eltern* (Abb. 5.1) von bis zu $\mathcal{O}(\log(m))$ anderen Knoten, und werden, jedes Mal wenn sie benötigt werden, aus dem Datenarray gelesen. Da sie, besonders bei den groben Verfeinerungen, auch im Speicher sehr weit von ihren hierarchischen Kindern entfernt sein können, verletzen diese Zugriffe das Prinzip der räumlichen Lokalität. Die

wiederholten Zugriffe können dabei auch zeitlich weit auseinander liegen, was die zeitliche Lokalität ebenso verletzt. Deshalb ist prinzipiell mit einer schlechten Cache-Ausnutzung zu rechnen.

Dies trifft allerdings nicht ganz zu: Durch die exponentielle Zunahme der Anzahl Knoten in den feineren Ebenen sind die Cache-Misses, die von den gröberen Ebenen verursacht werden, für die Gesamtlaufzeit des Algorithmus meist vernachlässigbar. Auf den feineren Ebenen werden die Werte nicht so oft eingelesen, und diese Zugriffe erfolgen dicht nacheinander. Außerdem werden die Knoten näherungsweise sequenziell abgearbeitet, was für die feineren Ebenen eine sehr gute Cache-Effizienz auch für diesen Algorithmus erlaubt.

5.2 Eindimensionaler Stack & Stream-Algorithmus

Die Nachteile des naiven Hierarchisierungsalgorithmus lassen sich mit einem Stack & Stream-Algorithmus umgehen. Dieser liest jeden Wert nur einmal ein, und stellt räumliche Lokalität durch die Verwendung von Stacks sicher. Besonders vorteilhaft ist, dass nur ein Stack der Größe $\mathcal{O}(\log m) = \mathcal{O}(l_{max})$ für ein Gitter mit l_{max} Verfeinerungen, benötigt wird. Der Speichermehraufwand ist also gering, und es ist unwahrscheinlich, dass bei Zugriffen auf den zusätzlichen Stack Cache-Misses auftreten.

Der Algorithmus basiert auf einer Grammatik, die die rekursive Struktur der Daten beschreibt, und einem dafür konstruierten Kellerautomaten. Ein Nachteil dieses Algorithmus ist, dass die Ausgabe nicht in der Reihenfolge geschrieben wird, in der die Eingabe gelesen wird; praktisch ist dies aber eine geringe Einschränkung, da bei der weiteren Verarbeitung ein ähnlicher Algorithmus genutzt werden kann, der die Umordnung rückgängig macht.

Der Algorithmus benutzt eine Grammatik, die nur ein Wort über einem unären Alphabet erzeugt:

$$\begin{aligned} A_{l_{max}} &\longrightarrow x\, A_{l_{max}-1}\, x \\ A_l &\longrightarrow A_{l-1}\, x\, A_{l-1}\, x\, A_{l-1} \\ A_0 &\longrightarrow x\, x \end{aligned} \tag{5.1}$$

mit dem Startsymbol $A_{l_{max}}$. Jeder Eingabewert wird durch ein Terminal x dargestellt, da die Werte zwar verschieden sein können, aber nicht unterscheidbar sind. Die erzeugte Sprache ist also trivial, daraus folgt aber nicht direkt, dass auch aus dieser Grammatik ein deterministischer Kellerautomat konstruiert werden kann. Hier ist das allerdings möglich, da sich die Ableitung aus der Grammatik ohne betrachten der Eingabe ergibt, ist die Grammatik in $LL(0)$. Da die Grammatik aber die Eingabe in sinnvolle Teile zerlegt, ist es möglich, mit den auf dem Stack eines LL-Parsers verfügbaren Werten die Hierarchisierung durchzuführen. Da die Terminale jedoch vom Stack entfernt werden müssen, wenn ihre Ableitung abgeschlossen ist, ergibt sich zwingend die Form der Ausgabe:

$$\begin{aligned} A_{l_{max}} &\longrightarrow A_{l_{max}-1}\, x\, x \\ A_l &\longrightarrow A_{l-1}\, A_{l-1}\, A_{l-1}\, x\, x \\ A_0 &\longrightarrow x\, x \end{aligned} \tag{5.2}$$

Interpretiert man die Ausgabe als Stack, so ergibt sich eine gespiegelte Form, da der Stack in umgekehrter Reihenfolge gelesen werden muss. Diese Form lässt sich auf die selbe Art und Weise parsen und in der ursprünglichen Form ausgeben.

Die Umsortierung, die durch den Algorithmus erfolgt, erinnert an die Umformung von der Infix- in die Postfix-Schreibweise eines mathematischen Ausdrucks. Aufgrund der Dreiteilung ergibt sich hier allerdings kein Baum, so dass die Bezeichnungen nicht direkt übernommen werden können. Es gibt zudem noch weitere, mögliche Darstellungen, z. B. die später eingeführte hierarchische Ordnung, die auf Binärbäumen nicht existieren. Der hier vorgestellte Algorithmus ähnelt sehr stark dem von Dijkstra eingeführten *Shunting yard* Algorithmus [Dij63] zur Konvertierung von ALGOL-Ausdrücken in Postfix-Form, ein frühes Beispiel für einen Stack & Stream Algorithmus.

Die Transformation lässt sich kompakt als Kombination der beiden Grammatiken darstellen. Die Grammatik wird dazu, zusätzlich zu den *Eingaberegeln*, die die Form der Eingabe beschreiben, mit *Ausgaberegeln* versehen. Die Ausgaberegeln werden hier mit einem Pfeil in umgekehrter Richtung dargestellt. Sie geben vor, wie die gelesenen Symbole wieder ausgegeben werden:

$$\begin{array}{lll} A_{l_{max}} & \longrightarrow x^{(1)}\, A_{l_{max}-1}\, x^{(2)} & \longleftarrow A_{l_{max}-1}\, x^{(1)}\, x^{(2)} \\ A_l & \longrightarrow A_{l-1}^{(1)}\, x^{(1)}\, A_{l-1}^{(2)}\, x^{(2)}\, A_{l-1}^{(3)} & \longleftarrow A_{l-1}^{(1)}\, A_{l-1}^{(2)}\, A_{l-1}^{(3)}\, x^{(1)}\, x^{(2)} \\ A_0 & \longrightarrow x^{(1)}\, x^{(2)} & \longleftarrow x^{(1)}\, x^{(2)} \end{array} \tag{5.3}$$

Die hochgestellten Indices dienen dabei zur eindeutigen Kennzeichnung und Zuordnung der Symbole auf der rechten Seite. Üblicherweise werden *LL*-Parser mit rekursivem Abstieg implementiert, wobei der Keller der Aufrufstack ist. Dann lässt sich eine solche Grammatik sehr leicht umsetzen, wenn die Nichtterminale auf Ein- und Ausgabe in der selben Reihenfolge auftauchen: die Nichtterminale werden durch Unterprogrammaufrufe des zur entsprechenden Regel gehörigen Unterprogramms ersetzt, die Terminale in der Reihenfolge der Eingaberegel vom Input gelesen und in lokalen Variablen gespeichert. Diese Werte werden wieder auf den Output geschrieben, wenn das entsprechende Terminal in der Ausgaberegel erscheint. In dieser Grammatik wird also im Startsymbol $A_{l_{max}}$ zunächst ein x eingelesen und einer lokalen Variable gespeichert. Dann wird A mit dem Parameter $l_{max} - 1$ aufgerufen, in diesem Aufruf wird der verbleibende Teilbaum eingelesen und wieder ausgegeben. Am Ende wird der andere Randwert x eingelesen und die beiden x wie in der Ausgaberegel gefordert ausgegeben. Das Unterprogramm A arbeitet analog nach der Regel für A_l, hier finden zwei zusätzliche rekursive Aufrufe am Anfang und Ende statt. Für den Sonderfall A_0 werden die rekursiven Aufrufe ausgelassen, wenn das höchste Level erreicht ist.

Offensichtlich ist diese Art von Implementierung nicht für jede Regel geeignet. Hier werden jedoch nur Grammatiken betrachtet, die sich so umsetzen lassen.

Für die Hierarchisierung ist diese Implementierungsform aber ungünstig, da die Daten aus weiter unten liegenden Stackframes nicht unmittelbar erreichbar sind. Deshalb wird hier ein expliziter Keller genutzt. Als Optimierung werden keine Nichtterminale als Platzhalter auf den Stack gelegt, wenn darauf keine Terminale folgen, da diese Informationen bei der

gegebenen Grammatik nicht zwingend erforderlich sind. Es bietet sich dann eine andere Erklärung des Algorithmus an: Wir betrachten die Werte der Eingabe, die mit einer *Höhe*, wie in Abb. 5.1, assoziiert sind. Diese Höhe stellt das Nichtterminal dar, aus dem der Wert abgeleitet wurde. Sie ist in der Eingabe nicht kodiert, aber beim Ablegen des Wertes auf dem Stack bekannt und wird auf dem Stack gespeichert. Zunächst wird nun, nach der ersten Regel, ein Wert eingelesen, der maximale Höhe hat. Die nächsten beiden Werte haben, aufgrund des rekursiven Abstiegs in der Grammatik, die minimale Höhe 1. Nun gilt für jeden eingelesenen Wert: wir nehmen zunächst an, dieser hat die geringste Höhe. Wenn aber die beiden letzten Werte auf dem Stack schon diese Höhe haben, dann muss die Höhe größer sein. Die beiden Werte vom Stack werden hierarchisiert, die Eltern sind der neue Wert und der auf dem Stack folgende, und werden auf die Ausgabe gelegt. Wir nehmen nun eine eins größere Höhe an und wiederholen den Vorgang, bis weniger als zwei Werte dieser Höhe auf dem Stack liegen. Dann wird der aktuelle Wert mit seiner dieser Höhe auf den Stack gelegt. Die Eingabe endet, wenn zwei Werte maximaler Höhe auf dem Stack liegen, dann werden diese noch ausgegeben.

Die Einträge auf dem Stack liegen immer der Höhe nach geordnet und in der selben Reihenfolge wie in der Eingabe auf dem Stack, zudem gibt es maximal zwei Einträge mit gleicher Höhe. Daraus folgt sofort, dass der Stack maximal $2(l_{max}+1)$ Einträge hat, also logarithmische Größe bezogen auf die Eingabe, und die Korrektheit ist leicht einzusehen. Ein *LL*-Parser erzeugt die selbe Anordnung der Terminale auf dem Stack, lagert auf dem Stack aber zusätzlich noch Nichtterminale, die hier durch das schrittweise Bestimmen der Höhe nicht benötigt werden.

Eine andere Anordnung der Knoten, die sich gut zur Stack & Stream-Hierarchisierung eignet, ist die *hierarchische* Ordnung. Diese Ordnung ergibt sich aus der Verwendung von Knotenstacks (Abschnitt 4.3.1 auf Seite 29), und ist anders als die gerade eingeführten Formen *symmetrisch*: nach der Traversierung liegen die Knoten zwar in einer anderen Reihenfolge wie zuvor, aber diese Reihenfolge kann vom selben Algorithmus benutzt werden, um eine Traversierung in umgekehrter Richtung durchzuführen. Anders als zuvor wird hier also nur ein Algorithmus benötigt. Während die vorher angenommene Form der Eingabe einer in-order Traversierung einer Baumes ähnelt, und die Ausgabe einer post-order Traversierung, ist die hierarchische Ordnung mit keiner üblichen Baumtraversierung vergleichbar. Die hierarchischen Abhängigkeiten bilden auch keine Baumstruktur, da jeder Knoten *zwei* hierarchische Eltern besitzt (vgl. Abb. 5.1 auf Seite 40).

Die Grammatik für die hierarchische Ordnung lautet wie folgt

$$\begin{array}{lll} A_{l_{max}} & \longrightarrow x^{(1)}\, x^{(2)}\, A_{l_{max}-1} & \longleftarrow A_{l_{max}-1}\, x^{(1)}\, x^{(2)} \\ A_l & \longrightarrow x^{(1)}\, A^{(1)}_{l-1}\, x^{(2)}\, A^{(2)}_{l-1}\, A^{(3)}_{l-1} & \longleftarrow A^{(1)}_{l-1}\, A^{(2)}_{l-1}\, x^{(1)}\, A^{(3)}_{l-1}\, x^{(2)} \\ A_0 & \longrightarrow x^{(1)}\, x^{(2)} & \longleftarrow x^{(1)}\, x^{(2)} \end{array} \tag{5.4}$$

und kann ebenfalls leicht als recursive-Descent implementiert werden. Obwohl aus Sicht der Knotenstacks für eine Dimension zwei Stacks erforderlich wären, kommt der Kellerautomat mit nur einem aus: Dieser Stack nimmt die Werte beider Knotenstacks abwechselnd auf, mit Platzhaltern für die Werte, die erst später gelesen werden.

Algorithmus 5.2 Cache-effiziente Hierarchisierung mit dem unidirektionalen Prinzip, für ein d-dimensionales Gitter mit m Knoten pro Dimension. HIER ist ein 1D-Hierarchisierungsalgorithmus, der m Knoten als Vektoren mit je s sequenziell im Speicher liegenden Einträgen bearbeitet.

```
procedure VEKTOR-HIER(data, m, d)
    s ← m^(d−1)
    HIER(data, m, s)
    if s > 1 then
        for i ∈ {1, ..., m} do
            VEKTOR-HIER(data[(i − 1) · s : i · s], m, d − 1)
        end for
    end if
end procedure
```

5.3 Unidirektionales Prinzip

Um auf mehrdimensionalen Gittern zu hierarchisieren können die eindimensionalen Algorithmen benutzt werden. Der naive Algorithmus lässt sich auch so erweitern, dass er direkt eine Hierarchisierung in d Dimensionen durchführt. Dann führt jedoch das Zugriffsmuster zu sehr schlechter Cache-Effizienz. Es gibt jedoch noch eine einfacher Lösung: die Hierarchisierung in d Dimensionen lässt sich zerlegen in d eindimensionale Hierarchisierungsoperationen, die in jeder der Dimensionen nacheinander und in beliebiger Reihenfolge ausgeführt werden können. Dieses Vorgehen wird als *unidirektionales Prinzip* bezeichnet.

Um ein volles Gitter in allen Dimensionen zu Hierarchisieren, muss also nur auf jeder eindimensionalen Zeile des Gitters die 1D-Hierarchisierung angewendet werden, anschließend auf jeder Spalte, usw., bis alle Dimensionen abgearbeitet sind. Aus Sicht der Cache-Effizienz ergibt sich hier allerdings ein Problem: bei einer zeilenweisen Ordnung der Daten ist ein z. B. spaltenweiser Zugriff für die Hierarchisierung in der 2. Dimension sehr ineffizient, da keine räumliche Lokalität gegeben ist. Für mehr Dimensionen verschärft sich das Problem entsprechend.

Eine potentielle Lösung ist der in Abschnitt 3.2.1 auf Seite 19 eingeführte Algorithmus zum Transponieren von Matrizen. Nach diesem Verfahren lässt sich die Ordnung der Daten vor der Hierarchisierung in einer Dimension so anpassen, dass die Hierarchisierungsalgorithmen selbst Cache-effizient sind. Die Anpassung selbst ist, durch diesen Algorithmus, ebenfalls Cache-effizient. Der entscheidende Nachteil ist, dass, auch wenn die Cache-Effizienz sehr hoch ist, die gesamten Daten für jede 1D-Hierarchisierung *und* für jede Umsortierung komplett aus dem Speicher gelesen und anschließend wieder dorthin geschrieben werden müssen. Die Geschwindigkeit dieser Operationen wird wahrscheinlich, auch bei optimaler Cache-Ausnutzung, durch die verfügbare Speicherbandbreite limitiert. Es wäre also schon deswegen sinnvoll, diese *Umsortierung* zu umgehen, und bestenfalls auch für die Hierarchisierung die Daten nicht jedes mal komplett neu einzulesen.

Die Umsortierung kann eingespart werden, in dem man die 1D-Hierarchisierungen aus einer anderen Sicht betrachtet. Bei einem d-dimensionalen Gitter mit m Knoten pro Dimension werden m^{d-1} Hierarchisierungen auf jeweils m Knoten benötigt. Da der Ablauf der Hierarchisierung jedoch nicht von den Daten abhängt und jedes dieser Teilprobleme die selbe Struktur hat, sind diese Hierarchisierungen äquivalent zu einer Hierarchisierung auf m Vektoren mit m^{d-1} Einträgen, die jeweils die Knoten einer kompletten $d-1$-dimensionalen Hyperebene umfassen. Alternativ sind auch andere Zerlegungen in niedriger dimensionale Hyperebenen möglich. Diese können nun immer so gewählt werden, dass die Hyperebenen und die Teilprobleme im Speicher sequenziell vorliegen. Die Vektoroperationen können dann in beliebiger Ordnung, z. B. sequenziell, erfolgen, was eine gute räumliche Lokalität gewährleistet. Da die Teilprobleme zusammenhängend im Speicher liegen, übertragen sich auch die räumlichen Lokalitätseigenschaften des verwendeten Algorithmus auf das Gesamtverfahren. Algorithmus 5.2 implementiert eine solche Zerlegung.

Weitere Speicherzugriffe können vermieden werden, indem bei Stack & Stream-Algorithmen der Output eines Algorithmus direkt mit dem Input des folgenden verbunden wird. Dadurch kann der Umweg über einen Stack, der möglicherweise in den Hauptspeicher geschrieben werden muss, vermieden werden. Dieser Ansatz wurde im Rahmen dieser Arbeit nicht verfolgt. Für eine Realisierung muss zum Beispiel auch betrachtet werden, dass die Anzahl Elemente der jeweils benutzten Vektoren, je nach Anzahl beteiligter Dimensionen sehr stark variiert, und dass eventuell andere Algorithmen verwendet werden sollten, wenn die Größe der Vektoren die Cache-Größe überschreitet.

6 Stack & Stream Algorithmus für dünne Gitter

Die Struktur von dünnen Gittern ist komplexer als die eines vollen Gitters. Anders als beim vollen Gitter gibt es keine offensichtliche, „natürliche“ Anordnung der Gitterpunkte, die auch als Anordnung im Speicher genutzt werden könnte. Der hier vorgestellte Algorithmus ist deshalb deutlich komplizierter als die bisher erwähnten; er nutzt die Eigenschaft eines dünnen Gitters, durch Überlagerung verschiedener voller Gitter entstanden zu sein. Der Anschaulichkeit halber wird im Folgenden meist der zweidimensionale Fall betrachtet; einige Aussagen behandeln aber auch d-dimensionale Gitter.

6.1 Unterraumschema

Das Unterraumschema stellt die hierarchische Struktur eines Gitters dar. Jede Zelle (x, y) kann als ein vollständiges Gitter mit x bzw. y Verfeinerungen in der entsprechenden Richtung aufgefasst werden. Ein volles Gitter mit n Verfeinerungen in jeder Dimension wird also durch die Zelle (n, n) dargestellt, zudem zeigt das Unterraumschema alle hierarchisch gröberen Gitter in den Zellen $\{0, \ldots, n\} \times \{0, \ldots, n\}$.

Ein dünnes Gitter entsteht nun durch Weglassen der Zellen (x, y) mit $x + y > n + 1$ (für zwei Dimensionen). Vom quadratischen Unterraumschema bleibt nun also eine dreieckige Hälfte. Da allerdings genau die Zellen mit hoher Verfeinerung und folglich vielen Freiheitsgraden weggelassen wurden, enthält das resultierende dünne Gitter sehr viel weniger Freiheitsgrade als ein volles Gitter mit vergleichbarer Verfeinerung (vgl. Abb. 2.2 auf Seite 14).

Für den im Folgenden behandelten Algorithmus ist es günstiger, die Zellen des Unterraumschemas anders zu interpretieren. Die zu den jeweiligen Zellen korrespondierenden vollen Gitter sind nicht disjunkt, jedes Gitter enthält die Freiheitsgrade aller hierarchisch gröberen Gitter. Dies kann vermieden werden, indem jede Zelle (x, y) als *Differenz* aufgefasst wird, die nur die Freiheitsgrade enthält, die in keinem gröberen Gitter auftauchen.

Nun stellt jedes Rechteck $(a, b) - (c, d)$, definiert durch die linke untere Ecke (c, d) und die rechte obere Ecke (a, b), aus Zellen im Unterraumschema, ein Gitter dar:

- Wenn das Rechteck im Ursprung gewurzelt ist, also jeden der d Ränder des Schemas berührt, handelt es sich um ein volles Gitter.
- Berührt das Rechteck nur $d - 1$ Ränder, im Zweidimensionalen also eine Zeile oder Spalte, so handelt es sich um eine Differenz zwischen zwei vollen Gittern, die durch Vergröbern eines vollen Gitters in einer Dimension entstanden ist.

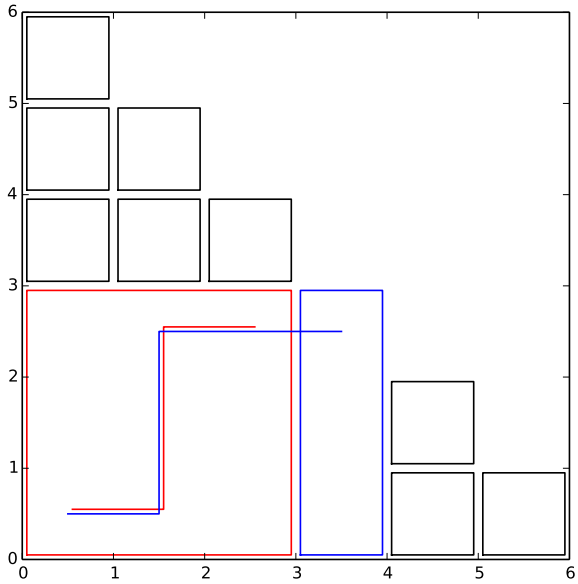

Abbildung 6.1: Ein Unterraumschema mit dem vollen Gitter $(3,3)-(0,0)$ mit Anordnung 1221 und der Gitterdifferenz $(4,3)-(3,0)$ in Anordnung 12211.

- Berührt das Rechteck nur $d-k$ Ränder, so ist es eine k-fache Differenz, die durch Vergröbern einer $k-1$-fachen Differenz in einer anderen Dimension entstanden ist.
- Insbesondere ist jede einzelne Zelle eine d-fache Differenz. Die Zelle (x,y) (Rechteck $(x+1,y+1)-(x,y)$) enthält genau die Freiheitsgrade, die nötig sind um aus den vollen Gittern $(x-1,y)-(0,0)$ und $(x,y+1)-(0,0)$ das volle Gitter $(x,y)-(0,0)$ zu bilden.

Später werden die Freiheitsgrade einer Zelle oder eines Rechtecks mit Hilfe einer raumfüllenden Kurve geordnet und auf verschiedenen Stacks abgelegt. Das Unterraumschema wird so zur Datenstruktur.

6.1.1 Verfeinerungen und Anordnungen

Um die Gitter auf Stacks ablegen zu können, muss die mehrdimensionale Struktur der Gitter auf die eindimensionale Struktur eines Stacks abgebildet werden. Eine naive, zeilenweise Anordnung wäre möglich, eine solche macht es aber schwierig die Freiheitsgrade verschiedener Gitter im Sinne eines Stack & Stream-Verfahrens zu kombinieren. Deshalb werden raumfüllende Kurven, genauer Peanokurven, benutzt. Die Gitter werden wie in Abschnitt 4.3 auf Seite 28 traversiert, und die Freiheitsgrade so abgelegt wie sie von der Traversierung auf den Output geschrieben bzw. vom Input gelesen werden.

Da die Gitter meist verschiedene Verfeinerungen in verschiedenen Dimensionen aufweisen, kann die übliche abwechselnde Verfeinerung der Peanokurve nicht genutzt werden. Stattdes-

sen wird jedem Gitter eine spezifische Verfeinerungsfolge zugeordnet, die eine Peanokurve definiert, welche wiederum die Anordnung der Knoten bestimmt.

Eine solche *Anordnung* kann als String dargestellt werden, in dem jedes Zeichen die Dimension angibt, in die verfeinert wird. Eine Verfeinerungsfolge für ein Gitter $(x, y) - (c, d)$ muss genau x mal 1 und y mal 2 enthalten, damit die richtigen Knoten traversiert werden, es ist aber jede beliebige Anordnung dieser Schritte möglich. Für die später eingeführten Algorithmen sind die Anordnungen meist *sortiert*, d. h. Verfeinerungen in der selben Dimension erfolgen direkt nacheinander. Dann kann sowohl die flache als auch die getrennte Speicherung problemlos benutzt werden. Sofern nicht anders angegeben werden volle Gitter getrennt, Differenzen aber immer in flach abgelegt. Praktisch muss neben der Anordnung noch betrachtet werden, ob die Freiheitsgrade in ursprünglicher oder gespiegelter Reihenfolge vorliegen, also ob bisher eine gerade oder ungerade Anzahl Vollgitter-Traversierungen durchgeführt wurde. Dies lässt sich, falls die falsche Reihenfolge vorliegt, immer durch eine zusätzliche Traversierung beheben und wird hier nicht näher betrachtet.

Im Unterraumschema kann eine Anordnung als Pfad dargestellt werden, der in Zelle $(0, 0)$ beginnt und in der oberen rechten Ecke, also (x, y), endet. Auch für die nicht-vollen Gitter ist die Anordnung ein solcher Pfad, d. h. die Anordnung einer Gitterdifferenz wird durch die Anordnung des entsprechenden vollen Gitters bestimmt.

Im Folgenden werden Operationen eingeführt, die die Verfeinerung eines vollen Gitters verändern, indem sie die Freiheitsgrade einer Verfeinerung aus einer anderen Quelle lesen oder auf einen anderen Stack schreiben. Bei solchen Operationen muss aber auch immer die Anordnung beachtet werden, zudem wird eine Operation eingeführt, die die Anordnung unter bestimmten Umständen verändert. Generell sind Veränderungen an der Anordnung aber nur schwierig als Stack & Stream Verfahren zu realisieren.

6.2 Operationen

Für den Dünngitteralgorithmus werden Operationen benötigt, die es erlauben, ein volles Gitter so zu verfeinern oder zu vergröbern, dass alle Teilgitter des dünnen Gitters erreicht werden können. Dann kann, an geeigneten Stellen, die Hierarchisierung ausgeführt werden, und nach Traversierung aller im dünnen Gitter enthaltenen Freiheitsgrade ist das dünne Gitter vollständig hierarchisiert. Alle hier vorgestellten Operationen sind, durch die Benutzung von Stacks, invertierbar. Die Algorithmen zur Realisierung der Operationen enthalten keine datenabhängigen Entscheidungen, d. h. man erhält die inverse Operation ganz einfach durch rückwärts ablaufen lassen des Programms und Vertauschen von *push* und *pop*.

Traversierung. Alle Freiheitsgrade und die dadurch aufgespannten Zellen eines vollen Gitters werden traversiert, dabei können Funktionen ausgewertet werden. Die Traversierung liest von einem Stack und schreibt die Knoten in der selben Anordnung auf einen anderen Stack. Wie oben erwähnt ist die Reihenfolge nach der Traversierung umgekehrt, dies gilt auch für alle folgenden Operationen. Die Traversierung ist aber zu sich selbst invers und kann

auch genutzt werden um die Reihenfolge umzukehren, falls dies für eine andere Operation notwendig ist.

Vergröberung (Verfeinerung). Wie bei der Traversierung werden die Knoten gelesen (geschrieben). Es werden jedoch zwei Stacks geschrieben (gelesen): ein Stack enthält die Knoten des verbleibenden Gitters, der andere die Knoten der einfachen Differenz in Richtung der letzten Verfeinerung. Die Operationen sind invers zueinander, sie verändern das Gitter um eine einfache Differenz und verändern die Anordnung nur an der letzten Stelle. Die Vergröberung/Verfeinerung bietet eine günstige Gelegenheit zum hierarchisieren/dehierarchisieren der Differenz in der entsprechenden Dimension, da die Freiheitsgrade hier später nicht mehr benötigt werden.

Alternative Vergröberung (Verfeinerung). Eine Vergröberung (Verfeinerung) des Gitters ist auch in einer anderen Dimension als der Letzten der Anordnung möglich, solange es sich um die letzte Verfeinerung in dieser Dimension in der Anordnung handelt (Abb. 6.3). Eine Hierarchisierung ist hier im Allgemeinen nicht möglich. Es bietet sich aber die Möglichkeit die Differenz noch einmal in einer anderen Dimension zu Vergröbern und so zusätzlich mehrfache Differenzen zu erzeugen. Eine detaillierte Beschreibung folgt in Abschnitt 6.2.1.

Umsortierung. Diese Operation lässt das Gitter unverändert, verändert aber die Anordnung. Voraussetzung ist, dass eine einzelne Verfeinerung in einer Richtung am Ende steht: dann wird diese durch die vorhergehenden Verfeinerungen in einer Richtung hindurch nach vorn geschoben, wie in Abb. 6.4 dargestellt. Wie zuvor angedeutet ist diese Operation deutlich aufwändiger zu realisieren als die übrigen Operationen, sie wird gesondert in Abschnitt 6.2.2 erklärt.

6.2.1 Die alternative Verfeinerung/Vergröberung

Die Vorgehensweise der normalen Verfeinerung wurde bei der Einführung schon kurz beschrieben: Im Grunde wird eine Traversierung des feineren Gitters durchgeführt, mit dem einzigen Unterschied, dass in den letzten Rekursionsschritten der rekursiven Verfeinerung, also in den Blättern des Aufrufbaumes, ein anderer Stack als Input genutzt wird. Statt dem Stack, der normalerweise für die Speicherung des vollen Gitters benutzt wird, wird hier ein Stack gelesen, der die Differenz enthält. Die Vorgehensweisen für Verfeinerung und Vergröberung sind genau identisch, da es sich um zueinander inverse Operationen handelt.

Um die alternative Verfeinerung zu verstehen, hilft ein näherer Blick auf den Aufrufbaum. Abb. 6.5 zeigt die rekursiven Aufrufe der Grammatik-Implementierung für die Peanokurve und die im jeweiligen Teilbaum eingelesenen Werte. Für die normale Verfeinerung würden die Freiheitsgrade abgetrennt, die auf Blattebene hinzukommen: Eine 1D-Verfeinerung in jeder Spalte. Für die alternative Verfeinerung müssen nun *ganze Spalten* abgetrennt werden, und zwar genau die Spalten, die in der letzten 1-Verfeinerung hinzukamen. Diese zu isolieren

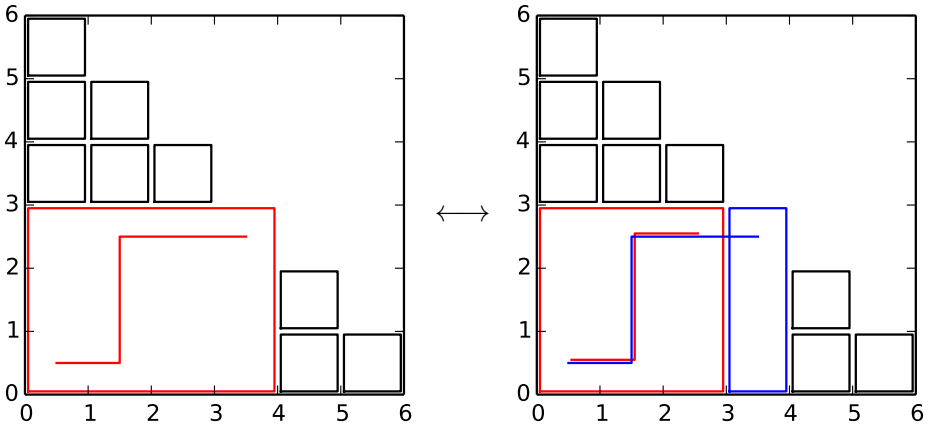

Abbildung 6.2: Die normale Verfeinerung/Vergröberung zwischen 12211 und 1221.

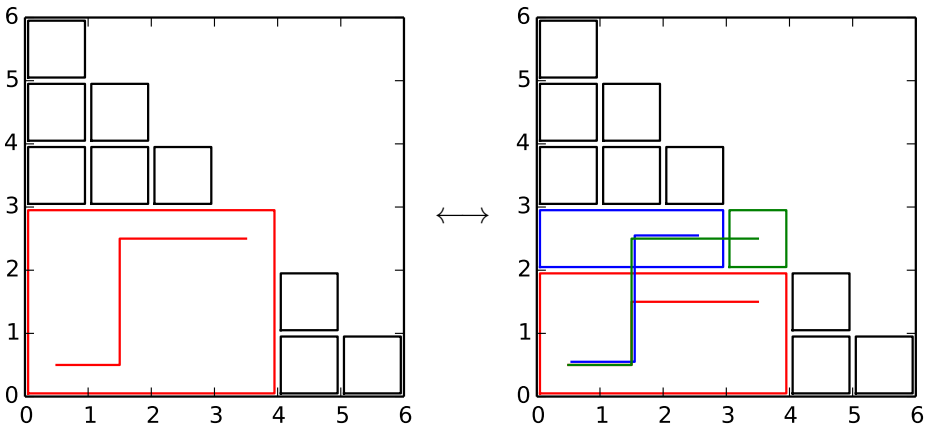

Abbildung 6.3: Die alternative Verfeinerung/Vergröberung zwischen 12211 und 1211.

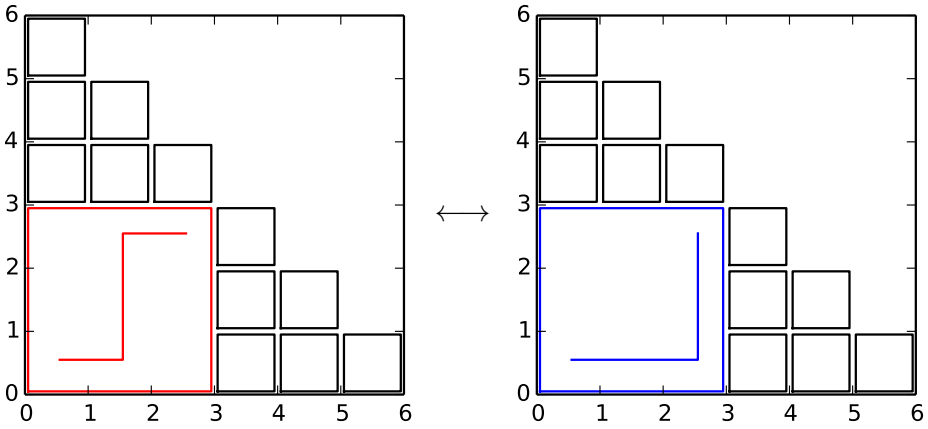

Abbildung 6.4: Die *Umsortieren*-Operation zwischen 1221 und 1122.

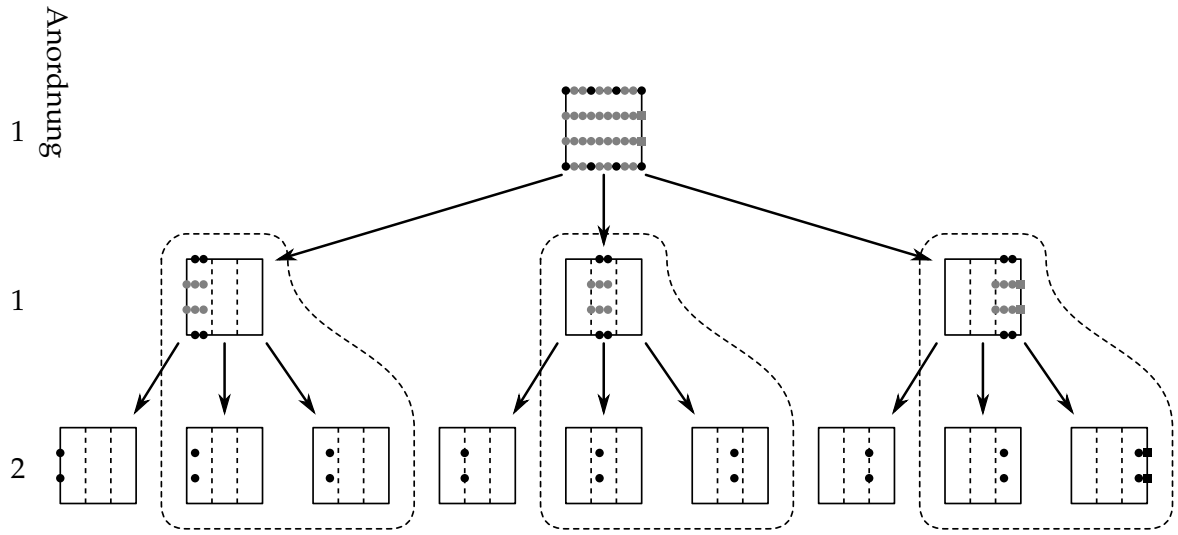

Abbildung 6.5: Aufrufbaum für die alternative Vergröberung von 112 in 1-Richtung. Die Knoten in den umrandeten Teilbäumen werden entfernt, mit Ausnahme der eckig dargestellten Knoten am rechten Rand, die aufgrund der getrennten Speicherung nicht auf dem Output landen.

wird dadurch vereinfacht, dass eine *getrennte* Speicherung der Freiheitsgrade vorausgesetzt werden kann: Dann wird bei der Verfeinerung genau eine Spalte pro Aufruf vom Input gelesen, bzw. bei der Vergröberung genau eine Spalte pro Aufruf geschrieben. Es müssen also nur die Spalten, die im 2. und 3. rekursiven Aufruf erzeugt werden, isoliert behandelt werden. Es genügt, bei diesem Aufruf den Input/Output zu verändern. Wenn diese Änderung dann rekursiv weitergegeben wird, dann wird genau die gewünschte Differenz isoliert. Die Eigenschaften der Peanokurve gewährleisten, dass die verbleibenden Knoten vorher wie nachher in der selben Reihenfolge bearbeitet werden, so dass dort keine Änderung nötig ist. Die potenziell problematische Randbehandlung wird durch die getrennte Speicherung vermieden.

Nach der alternativen Verfeinerung/Vergröberung können auf der Differenz und/oder auf dem vollen Gitter weitere Verfeinerungen/Vergröberungen in anderen Dimensionen stattfinden. Im zweidimensionalen Fall ist die zweite Operation immer eine normale Verfeinerung/Vergröberung, bei der dann hierarchisiert werden kann. Im höherdimensionalen Fall können auch mehrere alternative Operationen aufeinander folgen, dies ist sogar notwendig, um d-fache Differenzen, also einzelne Zellen des Unterraumschemas, zu isolieren. Während einer solchen Operation treten dann verschiedene, $d-1,\ldots,1$-fache, Differenzen auf, es werden also d zusätzliche Inputs/Outputs für die Operation benötigt. Algorithmus 6.1 skizziert die realisierte Implementierung für zwei Dimensionen.

Die normale Vergröberung kann als Sonderfall der alternativen Vergröberung angesehen werden, wenn die Vergröberungsrichtung die der letzten Verfeinerung ist. Es ist allerdings sinnvoll diese Operationen zu trennen, da bei der normalen Vergröberung eine Hierarchisierung möglich ist, bei der alternativen Vergröberung hingegen nicht: Die hierarchischen

Algorithmus 6.1 Die alternative Vergröberung für zwei Dimensionen. Es wird eine zweifache Differenz, also eine Zelle im Unterraumschema abgespalten (Abb. 6.3). P ist das Unterprogramm, das die Grammatik für die Traversierung realisiert. Nach Abarbeitung der übergebenen Anordnung *ref* wird ein Callback aufgerufen, der Parameter *segment* $\in \{1,2,3\}$ gibt an, welcher Teilbereich der aktuellen Verfeinerung gerade bearbeitet wird.

```
procedure ALT-COARSEN(dim, out1, out2)
        // Zerlegen des Anordnungs-Strings, praktisch z. B. mit regulärem Ausdruck
    ref1 d1 ref2 d2 ← Anordnung    mit d1 = dim ≠ d2, |d2| = 1 und keinem dim in ref2
    procedure COARSEN1(...)
        P(ref = d1, Output = out1, callback = COARSEN2, ...)
    end procedure
    procedure COARSEN2(segment, ...)
        if segment = 1 then
            P(ref = ref2 d2, Output = default, ...)
        else
            P(ref = ref2, Output = out1, callback = COARSEN3, ...)
        end if
    end procedure
    procedure COARSEN3(...)
        P(ref = d2, Output = out2, ...)
    end procedure
    Traversierung vorbereiten
    P(ref = ref1, callback = COARSEN1, ...)
    Traversierung nachbereiten
    Anordnung ← ref1, ref2, d2
end procedure
```

Eltern liegen in anderen Teilbäumen, die in anderer Reihenfolge traversiert werden, und sind damit nicht erreichbar.

Prinzipiell ist es auch möglich zuerst eine normale Verfeinerung und anschließend, auf der Differenz, eine alternative Verfeinerung zu realisieren, die Implementierung ist dann allerdings etwas aufwändiger.

6.2.2 Die Umsortier-Operation

Die Umsortier-Operation ändert die Anordnung der Knoten, aber nicht das Gitter. Im Allgemeinen muss für eine solche Operation auf Knoten zugegriffen werden, die nicht lokal verfügbar sind, also nicht in geringem Abstand auf den Stacks zu liegen kommen, und es ist nicht möglich die Operation als Stack & Stream-Verfahren zu implementieren. Für die hier

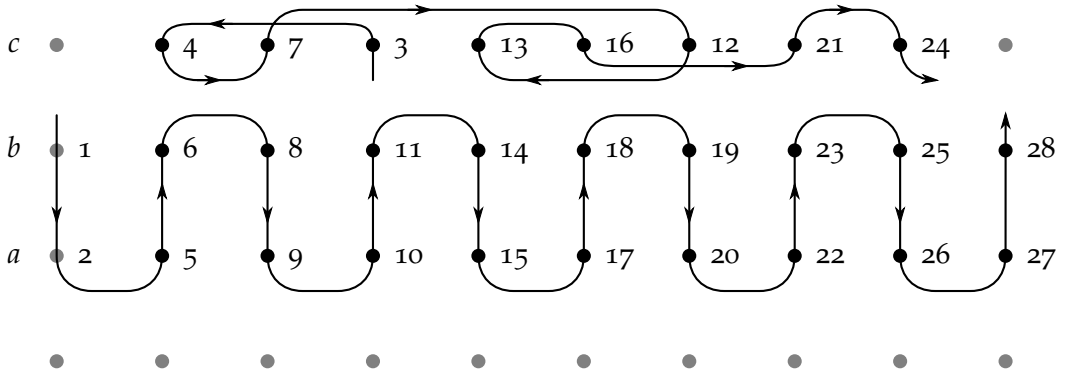

Abbildung 6.6: Anordnung der Knoten im 1. Teilbereich beim Umsortieren von 2112 nach 2211. Die inneren Knoten (Zeilen *a*, *b*) liegen mäanderförmig, die am Rand (Zeile *c*) jedoch hierarchisch geordnet dazwischen. Die grauen Knoten werden in anderen Zellen behandelt, Sonderfälle links sind schon an die Struktur angepasst.

betrachtete Operation ist dies allerdings dennoch möglich. Betrachtet wird eine Veränderung wie in Abb. 6.4. Dies kann ähnlich wie in Alg. 6.1 formalisiert werden:

$$\begin{array}{lll} \textit{ref1 ref2 dim} & \leftarrow \textit{Anordnung} & \text{mit } |\textit{dim}| = 1 \text{ und kein } \textit{dim} \text{ in } \textit{ref2} \\ \textit{Anordnung} & \leftarrow \textit{ref1 dim ref2} & \end{array} \tag{6.1}$$

Entscheidend ist, dass nur eine Verfeinerung in einer Dimension mit beliebig vielen anderen Verfeinerungen vertauscht wird. Da nur ein Verfeinerungsschritt betrachtet wird, müssen nur drei Unterteilungen in der Richtung, im Folgenden als Zeilen bezeichnet, gleichzeitig betrachtet werden.

Alle Verfeinerungsschritte in *ref1* sind für die Operation unbedeutend, da sie vorher wie nachher in der selben Reihenfolge abgearbeitet werden. Wir betrachten also nun eine Zelle des Gitters zu *ref1*. Im zweidimensionalen Fall und mit $dim = 2$, wie hier als Beispiel gewählt, kann *ref2* nur eine Folge von 1-Verfeinerungen sein, da nur 1- und 2-Verfeinerungen möglich sind. Wir nehmen an, dass zudem keine 1-Verfeinerung in *ref1* auftaucht. Dies ist für die später vorgestellten Algorithmen gegeben und vereinfacht die Sonderbehandlung der Randknoten. In der Anordnung auf dem Input werden die in der Zelle enthaltenen Gitterpunkte mäanderförmig, wie in Abb. 6.6 dargestellt, durchlaufen. Die Randzeile *c* ist in hierarchischer Anordnung dazwischen abgelegt. Die zwei Knoten am linken Rand liegen, abhängig davon ob eine flache oder getrennte Speicherung benutzt wird, anders vor; diese werden vorher in die Reihenfolge wie in Abb. 6.6 gebracht. Für den Output wird eine Anordnung angestrebt, die die Knoten zeilenweise abarbeitet. Dazu werden zunächst die Knoten innerhalb der Zelle umsortiert und anschließend eine Traversierung mit der neuen Anordnung durchgeführt, die auf dem Output die richtige Anordnung erzeugt.

Um dies zu realisieren, wird für jede Zeile ein Stack benutzt, der beim Einlesen vom Input die Elemente dieser Zeile aufnimmt und bei der folgenden Traversierung der Zeile selbst als Input dient. Da Input-, Output- und Transportstacks benutzt werden können sind

keine zusätzlichen Stacks nötig. Es stellen sich jedoch weitere Probleme: Einerseits sind auf dem Input zwischen den Freiheitsgraden, die umsortiert werden müssen, auch noch die Freiheitsgrade der Zeile am dem Rand der Zelle (c in Abb. 6.6), deren Reihenfolge unverändert bleiben muss, die aber beim Einlesen beachtet und zwischengespeichert werden müssen. Andererseits befinden sich die Freiheitsgrade im Inneren der Zelle auf dem Input in räumlich sequenzieller Anordnung, während sie für die Traversierung in hierarchischer Anordnung, wie in Abschnitt 5.2 auf Seite 41 eingeführt, benötigt werden. In hierarchischer Anordnung befinden sich auch die Freiheitsgrade der Randzeile. Weitere Sonderfälle sind die Knoten an den kurzen Randseiten, die je nach dem ob eine flache oder getrennte Speicherung gewählt wurde anders liegen oder auf einem anderen Stack liegen. Da es sich dabei aber um nur vier Knoten handelt, können diese leicht angepasst werden.

Das Trennen der Werte vom Input und Umsortieren in die hierarchische Anordnung stellt nun kein prinzipielles Problem mehr dar, ist aber technisch eine Herausforderung. Für die Implementierung wurde das Einlesen und Umsortieren mit einem LL-Parser mit Ausgaberegeln realisiert:

$$
\begin{array}{lll}
S_k & \longrightarrow b^{(1)}\, a^{(1)}\, Q_k\, a^{(2)}\, b^{(2)} & \longleftarrow \quad Q_k\, b^{(1)}\, b^{(2)}, \\
 & & \quad\; Q_k\, a^{(1)}\, a^{(2)}, \\
 & & \quad\; Q_k \\
Q_k & \longrightarrow c^{(1)}\, P_{k-1}^{(1)}\, a^{(1)}\, b^{(1)}\, c^{(2)}\, Q_{k-1}\, b^{(2)}\, a^{(2)}\, P_{k-1}^{(2)} & \longleftarrow \quad P_{k-1}^{(1)}\, Q_{k-1}\, b^{(1)}\, P_{k-1}^{(2)}\, b^{(2)}, \\
 & & \quad\; P_{k-1}^{(1)}\, Q_{k-1}\, a^{(1)}\, P_{k-1}^{(2)}\, a^{(2)}, \\
 & & \quad\; c^{(1)}\, P_{k-1}^{(1)}\, c^{(2)}\, Q_{k-1}\, P_{k-1}^{(2)} \\
P_k & \longrightarrow c^{(1)}\, Q_{k-1}^{(1)}\, a^{(1)}\, b^{(1)}\, c^{(2)}\, P_{k-1}\, b^{(2)}\, a^{(2)}\, Q_{k-1}^{(2)} & \longleftarrow \quad Q_{k-1}^{(1)}\, P_{k-1}\, b^{(1)}\, Q_{k-1}^{(2)}\, b^{(2)}, \\
 & & \quad\; Q_{k-1}^{(1)}\, P_{k-1}\, a^{(1)}\, Q_{k-1}^{(2)}\, a^{(2)}, \\
 & & \quad\; c^{(1)}\, Q_{k-1}^{(1)}\, c^{(2)}\, P_{k-1}\, Q_{k-1}^{(2)} \\
P_0 & \longrightarrow \varepsilon & \longleftarrow \quad \varepsilon, \varepsilon, \varepsilon \\
Q_0 & \longrightarrow \varepsilon & \longleftarrow \quad \varepsilon, \varepsilon, \varepsilon
\end{array}
\tag{6.2}
$$

Dabei ist das Startsymbol S_k, wobei k die Anzahl folgender 1-Verfeinerungen ist. Die Terminale a, b und c bezeichnen tatsächlich das selbe Terminalzeichen, da die Werte auf dem Input nicht unterschieden werden können, die verschiedenen Bezeichnungen erleichtern aber das Verständnis. Die drei Ausgaberegeln für jede Regel beschreiben die drei Ausgabestacks für die Zeilen a, b und c.

In der Grammatik ist jedoch die Reihenfolge der Knoten in jeder Zeile festgelegt, und für die folgende Traversierung liegt eine Zeile in der verkehrten Reihenfolge. Diese kann jedoch in mit Durchlauf mit einem Parser für die hierarchische Anordnung in 1D nach (5.4) behoben werden. Bis auf vier Eckknoten, die für die getrennte Form auf einen separaten Stack müssen, liegen die Knoten dann richtig und können für die Traversierung von den jeweiligen Stacks gelesen werden.

Die inverse Operation zur Umsortierung wird hier nicht näher beschrieben, sie ergibt sich durch Umformen der Grammatik bzw. durch rückwärts Ablaufen lassen des selben Algorithmus. Für mehr als zwei Dimensionen lässt sich ein ähnliches Verfahren anwenden; solange nur eine Verfeinerung betrachtet wird, genügen drei Stacks für die Umsortierung. Es sind jedoch auch Operationen denkbar, die zwei Verfeinerungen in verschiedenen Richtungen behandeln.

6.3 Dünngitter-Traversierung

Um Berechnungen auf einem dünnen Gitter auszuführen, müssen nun alle Zellen des Unterraumschemas abgearbeitet werden. Dazu werden die oben eingeführten Operationen benutzt: Ausgehend von einem initialen, vollen Gitter, z. B. der Zelle $(0,0)$, werden sukzessive Verfeinerungen ausgeführt, bis alle Zellen des Unterraumschemas in das volle Gitter integriert und traversiert wurden. Dazwischen sind auch Vergröberungen notwendig, um nicht die Zellen integrieren zu müssen, die beim dünnen Gitter explizit weggelassen werden sollen. Dies ist entscheidend, da die Anzahl Freiheitsgrade, die zu jeder Zelle des Unterraumschemas gehören, nach rechts und nach oben hin exponentiell zunimmt.

Für die hier betrachtete Hierarchisierung ist es zudem bedeutend, *wie* eine Zelle im hierarchischen Abstieg betreten wird. Die Hierarchisierung ist nur in der Richtung möglich, in der die Zelle betreten wurde, bzw. in der sie verlassen wird. Eine Zelle kann nur dann in einer anderen Richtung verlassen werden, wenn zuvor eine Umsortierung stattfindet. Auf die oben eingeführten Operationen bezogen bedeutet das, dass ein Knoten mindestens einmal in 1- und in 2-Richtung von einer normalen Verfeinerung integriert bzw. in einer normalen Vergröberung abgetrennt werden muss. Diese Anforderung führt direkt zu einem ersten Algorithmus, der die Hierarchisierung in zwei Dimensionen bewerkstelligt.

6.3.1 Einfacher Hierarchisierungsalgorithmus

Die Grundidee dieses Algorithmus ist es, der *Diagonalen* im Unterraumschema, also den Gittern mit der stärksten Verfeinerung $x + y = n + 1$, zu folgen, und diese vollen Gitter nacheinander zu bearbeiten. Vor jeder Verfeinerung in einer Richtung muss eine Vergröberung in der anderen Richtung erfolgen, damit das neue, volle Gitter nicht über die Diagonale herausragt.

Wir beginnen also mit der Zeile $(n+2,1) - (0,0)$. Die Freiheitsgrade dieser Zeile werden vom Input gelesen, eine Hierarchisierung in 2-Richtung ist nicht erforderlich, da es sich um die Freiheitsgrade am Rand handelt. Dieses Gitter wird nun mit der normalen Vergröberung in 1-Richtung vergröbert, die dabei abgetrennten Freiheitsgrade werden in 1-Richtung hierarchisiert und auf den Output geschrieben. Dann wird eine normale Verfeinerung in 2-Richtung vom Input gelesen, und während des Verfeinerns hierarchisiert. Dabei müssen allerdings die ursprünglichen Funktionswerte weiterhin mitgeführt werden, weil diese später

Algorithmus 6.2 Der einfache Hierarchisierungsalgorithmus.

```
procedure SIMPLE-HIER(level, inp, out)
    TRAVERSE(Input = inp)
    for i ∈ {0, ..., level} do
        COARSEN(Output = out)                    // Hier in-place hierarchisieren
        REFINE(Input = inp)          // Hier hierarchisieren, Funktionswerte behalten
        REORDER
    end for
    TRAVERSE(Output = out)
end procedure
```

wieder für die Hierarchisierung der nächsten Zeile benötigt werden. Nun sollte eine Vergröberung in 1-Richtung folgen. Würde diese mit der alternativen Vergröberung durchgeführt, die eine Vergröberung hier erlauben würde, dann könnte nicht in 1-Richtung hierarchisiert werden. Deswegen wird zunächst umgeordnet und dann normal vergröbert und hierarchisiert. Dabei dürfen die originalen Funktionswerte bzw. in 2-Richtung hierarchisierten Werte überschrieben werden, da die Freiheitsgrade nun auf dem Output landen und nicht für weitere Hierarchisierungen benötigt werden. Dies kann sukzessive fortgesetzt werden, bis die Spalte $(1, n+2) - (0, 0)$ erreicht ist, schließlich wird diese ebenfalls auf den Output geschrieben und das Gitter ist komplett hierarchisiert. Algorithmus 6.2 und Abb. 6.7 stellen dieses Verfahren dar.

Wie in Abb. 6.7 zu erkennen, werden immer ganze Zeilen des Unterraumschemas eingelesen und ganze Spalten auf den Output geschrieben. Die Gitter, die auf dem Output zu liegen kommen, haben also genau die Form und Reihenfolge, die für einen weiteren Durchlauf von SIMPLE-HIER z. B. zur Dehierarchisierung benötigt werden. Tatsächlich ist es allerdings nicht möglich, die Ausgabe direkt als Eingabe für die selbe Traversierung zu benutzen, da die Differenzen auf dem Output eine Anordnung der Form 11122 haben, auf dem Input jedoch eine der Form 11221 benötigt wird. Es ist aber möglich, den selben Algorithmus rückwärts ablaufen zu lassen: Aus der Umsortierung wird dann die inverse Umsortierung, die genau die nötige Modifikation durchführt.

Der Algorithmus läuft in Linearzeit zur Anzahl Freiheitsgrade, obwohl viele Freiheitsgrade bei mehreren Operationen traversiert werden. Insgesamt finden $\mathcal{O}(n)$ Vollgitter-Operationen statt, die jeweils auf einem Gitter (x, y) mit $x + y = n + 1$ arbeiten, also $\mathcal{O}(3^x) \cdot \mathcal{O}(3^y) = \mathcal{O}(3^n)$ Knoten bearbeiten. Dies ist asymptotisch proportional zur Anzahl Freiheitsgrade im dünnen Gitter, $\mathcal{O}(3^n \cdot n) = \mathcal{O}(m \log m) = \mathcal{O}(N)$.

Eine Verallgemeinerung für mehr als zwei Dimensionen ist bei diesem Algorithmus schwierig, da die Diagonale bei mehr als zwei Dimensionen zu einer Hyperebene wird. Zudem ist es nicht mehr möglich, die Hierarchisierungsoperationen ausschließlich bei Ein- und Ausgabe durchzuführen, da nicht nur zwei, sondern d notwendig sind. Eine mögliche Erweiterung wird im nächsten Abschnitt eingeführt.

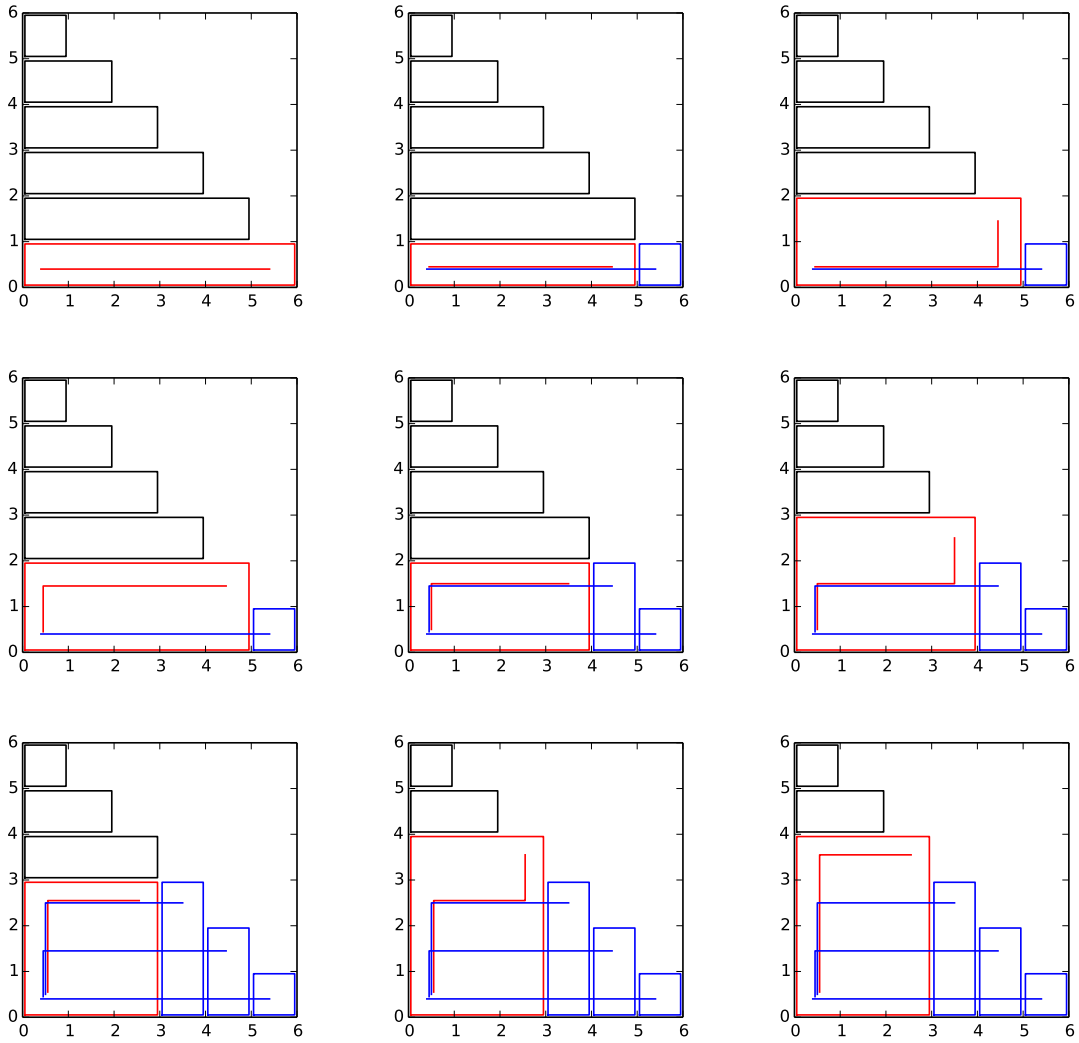

Abbildung 6.7: Die ersten Operationen des Algorithmus SIMPLE-HIER (Alg. 6.2) mit Level 5.

Praktische Realisierung

Alle hier vorgestellten Operationen wurden als Prototyp in Python realisiert. Für die Vollgitter-Traversierung anhand von Peanokurven wurde Grammatik 4.8 mit den zusätzlich nötigen Randbehandlungen wie in Kapitel 4 erläutert implementiert. Die Operationen konnten dann analog zum hier dargestellten Pseudocode umgesetzt werden. Für die Umsortierung wurde Grammatik 6.2 als rekursive-Descent Parser implementiert, zudem wurden die erwähnten Sonderbehandlungen ergänzt.

Mit den so implementierten Operationen kann dann Algorithmus 6.2 für die Dünngitter-Hierarchisierung wie dargestellt ausgeführt werden. Zusätzlich notwendig sind Parameter,

die für die Operationen angeben ob flache oder getrennte Speicherung benutzt wird, da manche Operationen nur für eine der beiden Formen implementiert sind. Außerdem muss das volle Gitter zu Beginn mit der richtigen Anordnung initialisiert werden.

Um den Inputstack nicht spezifisch für den Algorithmus initialisieren zu müssen, wurde stattdessen ein Generator benutzt, der beim lesen eines Knotens vom Input die Position des Gitterpunktes bestimmt und eine Testfunktion an dieser Stelle auswerte, um einen Funktionswert zu erhalten. Abb. 6.8 zeigt die so erhaltenen Funktionswerte aufgetragen an den Gitterpunkten, die Anordnung auf dem Inputstack ist nicht dargestellt. Nach Durchlauf des Hierarchisierungsalgorithmus ergeben sich die hierarchischen Überschüsse, die in Abb. 6.9 aufgetragen wurden.

Auffallend ist, dass, anders als in Kapitel 2 eingeführt, der Rand komplett durch Ansatzfunktionen vom Level 0 abgedeckt wird. Die in Kapitel 2 vorgestellte Form lässt sich aber ebenfalls leicht erreichen, in dem mit einem einen Schritt weniger verfeinerten Gitter begonnen wird und die erste Vergröberung sowie die letzte Verfeinerung ausgelassen werden.

6.3.2 Weitere Dünngitter-Traversierungen

Der einfache Hierarchisierungsalgorithmus zeigt noch einige Eigenschaften, die so nicht wünschenswert sind: Neben der Limitierung auf zwei Dimensionen ist es unpraktisch, dass vom Input ganze Zeilen/Spalten gelesen werden. Die Verarbeitung einzelner Zellen verspricht eine regelmäßigere Struktur, bei der nur während der Vergröberung hierarchisiert wird und so eine in-place Hierarchisierung möglich ist. Die eingeführten Operationen ermöglichen aber noch eine Vielzahl weiterer Dünngitter-Traversierungen, die auch für andere Berechnungen auf dünnen Gittern, nicht nur für die Hierarchisierung, geeignet sind. Im Folgenden werden weitere Verfahren vorgeschlagen, die sich mit den hier eingeführten Operationen umsetzen lassen, und andere Vor- und Nachteile haben.

Die Idee aus Abschnitt 6.3.1, der *Diagonale* zu folgen, lässt sich auf mehr Dimensionen verallgemeinern, indem diese wieder mäanderförmig durchlaufen wird. Dazu wird zum Vergröbern anstelle von Umsortierung und normaler Vergröberung die alternative Vergröberung benutzt, und die letzte Zelle jeder Spalte zusätzlich abgetrennt und auf den Output geschrieben. Die Spalten werden stattdessen nicht auf den Output, sondern auf einen temporären Stack geschrieben. Dann kann ein Schritt in 3-Richtung erfolgen und die Spalten vom temporären Stack, mit Hilfe der alternativen Verfeinerung kombiniert mit weiteren Freiheitsgraden vom Input, als Input für eine Traversierung auf der nächsten „Ebene“ im dreidimensionalen Unterraumschema genutzt werden. Auf dem Output liegen dann Stäbchen in 3-Richtung. Von diesen könnte aber wieder mit Hilfe der alternativen Vergröberung die jeweils letzte Zelle abgespalten werden, so dass die verbleibenden Stäbchen über einen weiteren temporären Stack für eine Traversierung nach dem nächsten Schritt in 4-Richtung zur Verfügung stehen.

Der Nachteil dieses Ansatzes liegt in der Verwendung der alternativen Vergröberung und Verfeinerung, die keine Hierarchisierung erlaubt. Eventuell müsste eine Umsortierung und inverse Umsortierung erfolgen, nur um die Freiheitsgrade dabei hierarchisieren zu können.

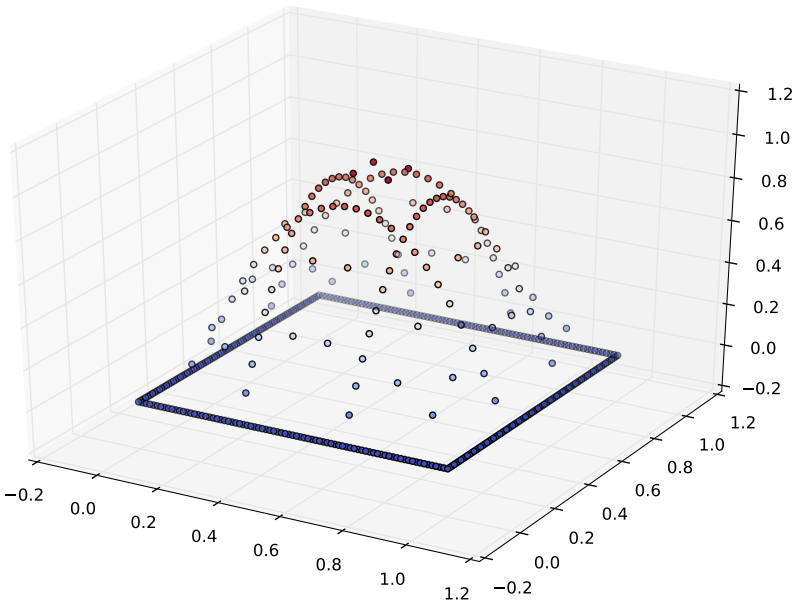

Abbildung 6.8: Die auf dem dünnen Gitter ausgewertete Testfunktion $f(\mathbf{x}) = \frac{1}{16}x_1(1 - x_1)x_2(1 - x_2)$.

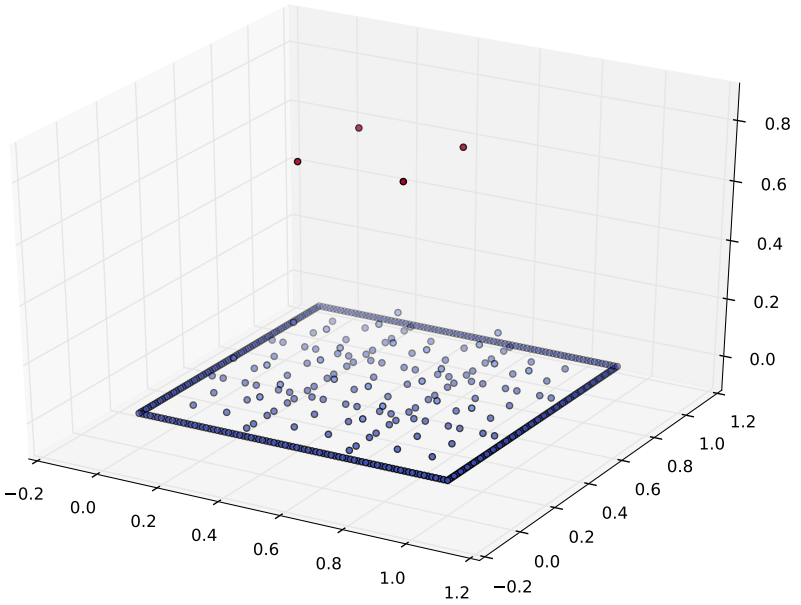

Abbildung 6.9: Die durch die Hierarchisierung in beiden Dimensionen erzeugen hierarchischen Überschüsse von $f(\mathbf{x})$.

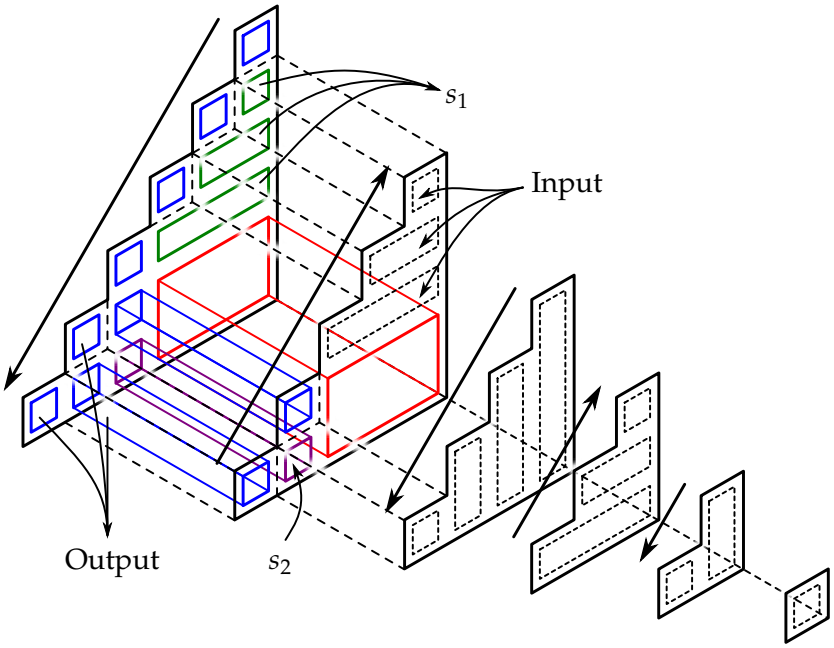

Abbildung 6.10: Verallgemeinerung der einfachen Dünngitter-Hierarchisierung für 3D. Gestrichelt sind die Differenzen, die noch auf dem Input liegen, die anderen Differenzen sind schon auf dem Output oder auf den temporären Stacks s_1 und s_2. Dargestellt ist die Situation nach dem Traversieren der 1. Ebene (links), wobei noch benötigte Werte s_1 geschrieben wurden, und den ersten Schritten auf der 2. Ebene, die noch benötigte Werte auf s_2 schreiben.

Ein anderer Ansatz ist ergibt sich, wenn man fordert, dass nur einzelne Zellen des Unterraumschemas auf In- und Output liegen. Beim Abspalten einer d-fachen Differenz kann immer in der letzten Dimension hierarchisiert werden. Durch Umsortieren kann dies immer eine andere Dimension sein, so dass am Ende vollständig hierarchisierte Werte bleiben. Um die Hierarchisierung in-place durchführen zu können, darf zudem nur beim Vergröbern hierarchisiert werden. In 2D muss also jede Zelle zweimal abgespalten werden, einmal vor dem Schreiben auf den Output, und einmal auf einen temporären Stack. Dass bei diesem Ansatz mehr Traversierungen notwendig sind, fällt wenig ins Gewicht, solange die Zellen auf der Diagonalen nicht zu oft besucht werden. Durch das exponentielle Wachstum der Anzahl Freiheitsgrade beim rekursiven Verfeinern befinden sich die meisten Freiheitsgrade in den Zellen der Diagonale.

Die eingeführten Operationen eröffnen viele Möglichkeiten zum Traversieren von dünnen Gittern mit Hilfe von Stack & Stream Algorithmen. Welcher Ansatz genau verwendet wird, muss von Anwendung zu Anwendung entschieden werden.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Cache-effizienter Stack & Stream Hierarchisierungsalgorithmus für dünne Gitter entwickelt. Dafür wurde eine Traversierung anhand einer Peanokurve genutzt, die eine Gitterstruktur auf Basis einer Dreiteilung benötigt, dafür allerdings die ausschließliche Verwendung von Stacks erlaubt und eine Verallgemeinerung für beliebig hohe Dimensionszahlen ermöglicht.

Im ersten Kapitel wurden die mathematischen Grundlagen zu Knotenbasis, hierarchischer Basis und dünnen Gittern eingeführt, die zum Verständnis der Hierarchisierung notwendig sind. Zudem wurden diese Grundlagen für beliebig t-geteilte Gitter erweitert, da später statt der üblichen 2-geteilten Gitter 3-geteilte benutzt werden.

Das folgende Kapitel führte das hier benutzte Cache-Modell auf Basis der Lokalitätsprinzipien ein. Es wurden zwei Ansätze zur Konstruktion Cache-effizienter Algorithmen vorgestellt: Beispielhaft für Divide & Conquer Algorithmen wurde der Cache-effiziente Algorithmus zur Matrix-Transponierung von Harald Prokop eingeführt, der auch als Bestandteil von Vollgitter-Hierarchisierungsalgorithmen dienen kann. Zusätzlich wurden Stack & Stream Algorithmen vorgeschlagen, die mit dem theoretischen Konzept von Kellerautomaten in Verbindung stehen.

In Kapitel 4 wurde die Peanokurve eingeführt und ein Schema vorgestellt, nach dem ein dreigeteiltes Gitter unter Betrachtung seiner hierarchischen Struktur anhand einer Peanokurve traversiert werden kann, so dass alle benötigten Daten über Stacks transportiert werden. Dieses Transportschema wird später die Basis für alle Algorithmen auf dünnen Gittern. Dieses ist nicht spezifisch für die Hierarchisierung, sondern könnte auch für andere Berechnungen genutzt werden.

Zur Hierarchisierung auf vollen Gittern wurden verschiedene Ansätze vorgestellt, die auf dem unidirektionalen Prinzip basieren. Neben dem klassischen, nicht Cache-effizienten Hierarchisierungsalgorithmus wurde ein Stack & Stream Verfahren für eine Dimension vorgestellt, das sich auch zu Cache-effizienten Algorithmen für höherdimensionale volle Gitter erweitern lässt.

Kernstück dieser Arbeit ist ein Stack & Stream Algorithmus zur Dünngitter-Hierarchisierung. Dieser wurde in Kapitel 6 vorgestellt und nutzt das Datentransportschema aus Kapitel 4 um die vollen Gitter, die in einem dünnen Gitter enthalten sind, zu traversieren. Dieser Ansatz ermöglicht eine Traversierung der komplexen Struktur eines dünnen Gitters mit einem Stack & Stream Verfahren, das Cache-Effizienz garantiert. Die vorgestellten Bausteine sind wieder generisch und nicht auf die Hierarchisierung beschränkt. Abschließend wurde ein Stack & Stream Hierarchisierungsalgorithmus für zweidimensionale dünne Gitter vorgestellt. Die

Eigenschaften der Peanokurve erlauben eine Verallgemeinerung aller Verfahren auf beliebige Dimensionszahlen, so dass auch dieser Algorithmus verallgemeinert werden kann, was jedoch die Komplexität noch weiter erhöht.

Insgesamt zeigte sich, dass die Peanokurve gut zur Realisierung von Stack & Stream Verfahren geeignet ist. Zwar müssen dreigeteilte Gitter benutzt werden, um die Peanokurve anwenden zu können, und mache Konzepte sind nicht leicht zu verstehen, dafür ist jedoch die Verallgemeinerung auf eine beliebige Dimensionszahl mit wenig zusätzlichem Aufwand und ohne weitere Sonderfälle möglich.

Ausblick

Die hier vorgestellten Ansätze zur Traversierung dünner Gitter bieten noch viel Potential, aber auch viel Bedarf an weiteren Entwicklungen. Der in dieser Arbeit eingeführte, einfache Hierarchisierungsalgorithmus weißt noch Beschränkungen auf, z. B. ist nicht vollständig geklärt wie eine Verallgemeinerung für d Dimensionen möglich ist.

Neben der Erweiterung auf höhere Dimensionszahlen ist auch eine Erweiterung auf andere Operationen als die Hierarchisierung denkbar: Einfache Stencil-Operationen, die mit den Eckknoten einer Zelle auskommen, können direkt umgesetzt werden, für Operationen, die Zugriff auf eine größere Nachbarschaft brauchen, ist eine Anpassung denkbar. In diesem Kontext können auch Hierarchisierungsoperationen auftauchen, bei denen nur einzelne Teilgitter nach Bedarf hierarchisiert werden. Hier verspricht der in dieser Arbeit kurz umrissene Ansatz, einzelne hierarchische Inkremente zu verarbeiten, Vorteile.

Ein anderer Aspekt ist die adaptive Verfeinerung des Gitters an Stellen, die eine höhere Auflösung benötigen. In diesem Zusammenhang wurden auch schon raumfüllende Kurven benutzt [Güno4], jedoch nicht im Verbindung mit dünnen Gittern. Die rekursive Verfeinerung einer Peanokurve ermöglicht eine lokale, adaptive Verfeinerung, dies gilt jedoch vor allem, wenn die üblichen, abwechselnden Verfeinerungen in verschiedenen Dimensionen benutzt werden. Für die in dieser Arbeit benutzten, sortierten Verfeinerungen ist nicht offensichtlich, wie eine adaptive Verfeinerung realisiert werden kann.

Ein anderes Feld, das hier ebenfalls nicht behandelt wurde, betrifft die praktische Umsetzung Cache-effizienter Algorithmen. In dieser Arbeit wurde die Cache-Effizienz ausschließlich anhand der abstrakten Lokalitätsprinzipien untersucht. Für eine konkrete Realisierung sind jedoch auch Anpassungen an die tatsächlich benutzten Cachestrukturen lohnend, z. B. der Wechsel auf alternative Algorithmen, wenn ein gewisse Problemgröße erreicht ist. Zudem bieten die hier vorgestellten Algorithmen noch Potential für Mikrooptimierungen, die die Anzahl benötigter Speicherzugriffe erheblich senken können. Für die Dünngitteralgorithmen ist beispielsweise die Kombination verschiedener Operationen in einer Traversierung denkbar, um die Gesamtzahl benötigter Traversierungen zu senken.

Schließlich wäre es für eine praktische Benutzung noch wünschenswert, auf Gittern mit einer Zweiteilung arbeiten zu können. Diese bieten durch eine geringere Anzahl Freiheitsgrade

praktische Vorteile. Mögliche Ansätze sind z. B. die Verwendung einer anderen raumfüllenden Kurve oder das Auslassen nicht verwendeter Zellen. Betrachtet man Abb. 4.2 auf Seite 24, so könnte man beispielsweise den mittleren Teilbereich auslassen, um so eine Z-Kurve zu erhalten, die ähnliche Eigenschaften wie die Peanokurve zeigt. Der Datentransport ist dann allerdings nicht mehr über Stacks möglich, stattdessen müssen Queues benutzt werden. Der andere Ansatz würde, bezogen auf Abb. 4.2, im dritten Teilbereich keine tatsächlichen Gitterzellen durchlaufen, sondern nur die Knoten am Rand noch einmal in umgekehrter Reihenfolge traversieren um die Knoten in die richtige Reihenfolge zu bringen.

Die Verwendung von Stack & Stream Algorithmen für dünne wie auch volle Gitter bietet viele Möglichkeiten, die in dieser Arbeit bei weitem nicht erschöpfend behandelt werden konnten. In den hier umrissenen Bereichen gibt es noch viel Raum für zukünftige Entwicklungen.

Literaturverzeichnis

[ASU86] A. Aho, R. Sethi, J. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986. (Zitiert auf Seite 22)

[Bad13] M. Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, Band 9 von *Texts in Computational Science and Engineering*. Springer-Verlag, 2013. (Zitiert auf den Seiten 23, 26 und 27)

[BG04] H.-J. Bungartz, M. Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004. (Zitiert auf den Seiten 9, 13 und 14)

[Dij63] E. Dijkstra. Making a Translator for ALGOL 60. *Annual review in automatic programming*, 3:347–356, 1963. (Zitiert auf Seite 42)

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, S. 285–297. IEEE, 1999. (Zitiert auf Seite 19)

[Gün04] F. Günther. *Eine cache-optimale Implementierung der Finite-Elemente-Methode*. Dissertation, Technische Universität München, Universitätsbibliothek, 2004. (Zitiert auf den Seiten 9 und 64)

[HP12] J. L. Hennessy, D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. (Zitiert auf den Seiten 17 und 18)

[Peh13] B. Peherstorfer. *Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques*. Dissertation, Department of Informatics, Technische Universität München, 2013. (Zitiert auf Seite 9)

[Pfl10] D. Pflüger. *Spatially adaptive sparse grids for high-dimensional problems*. Verlag Dr. Hut, 2010. (Zitiert auf Seite 9)

[Sch08] U. Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Hochschultaschenbücher. Spektrum Akademischer Verlag, 2008. (Zitiert auf Seite 22)

[Ste13] P. von Steht. *Cache-effiziente Hierarchisierung für dünne Gitter auf raumfüllenden Kurven*. Bachelorarbeit, Universität Stuttgart, 2013. (Zitiert auf Seite 9)

[Zum00] G. W. Zumbusch. A Sparse Grid PDE Solver. In H. P. Langtangen, A. M. Bruaset, E. Quak, Herausgeber, *Advances in Software Tools for Scientific Computing*, S. 133–177. Springer, Berlin, Germany, 2000. (Proceedings SciTools '98). (Zitiert auf Seite 9)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift